ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA DEL SOFTWARE

CONFIGURACIÓN Y USO DE PARALLELLA-16

CONFIGURATION AND USE OF PARALLELLA-16

Realizado por
Eliecer García Rey

Tutorizado por
Dr. D. Sergio Gálvez Roja

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, enero de 2015

Fecha defensa:

El Secretario del Tribunal

A mis padres

RESUMEN Y PALABRAS CLAVE

Parallella-16 es un ordenador de tipo SBC (del inglés *Single Board Computer*, traducido al español como *Ordenador de Placa Reducida*) con 16 núcleos fabricado por Adapteva. El precio de este ordenador, poco menos de 110€, y la capacidad de conectar varios de estos para trabajar como uno solo, lleva a sus responsables a utilizar como eslogan *"Supercomputing For Everyone"* (*Supercomputación para todo el mundo*).

Este trabajo fin de grado pretende ser un punto de inicio para todo aquel que quiera trabajar con Parallella-16, ya que su documentación está desactualizada, la descripción de sus librerías es pobre y carece de ejemplos sencillos y didácticos. Esto es evidente en su foro oficial, donde mucha gente pregunta cosas que podrían calificarse de básicas.

En este documento se explica la forma de configurar y trabajar con Parallella-16, incluyendo comentarios y opiniones de quien ha empezado a trabajar con ella sin ningún conocimiento. Tiene un capítulo de introducción a la programación en esta arquitectura utilizando el ejemplo *hello-world* que acompaña a su kit de desarrollo, la traducción de sus librerías y ejemplos sencillos que permiten conocer cómo utilizar las funciones de estas librerías.

Palabras clave: Informática, Supercomputación, Programación Paralela, Adapteva, Epiphany, Parallella-16, SBC (Single Board Computer), Multi-núcleo.

ABSTRACT AND KEYWORDS

Parallella-16 is a SBC (Single Board Computer) with 16 cores manufactured by Adapteva. The price of this computer, just under $120, and the ability to connect several of these to work as one, takes its responsibility to use the slogan *“Supercomputing for Everyone”*.

This work aims to be a starting point for anyone who wants to work with Parallella-16. The documentation of this project (Parallella-16) is outdated, the description of its libraries are poor and hasn’t simple and didactic examples. This is evident in his official forum where many people question things that could be described as basic.

This document describes how to configure and work with Parallella-16, including comments and opinions of those who have begun to work with it without any knowledge. It has a chapter of introduction to programming in this architecture using the *hello-world* example that came with your software kit development, translation of its libraries to Spanish and simple examples to learn how to use the functions of these libraries.

Keywords: Computing, Supercomputing, Parallel computing, Adapteva, Epiphany, Parallella-16, SBC (Single Board Computer), Multicore.

ÍNDICE

1 INTRODUCCIÓN.....11

2 PARALLELLA-16.....13
2.1 Adapteva.....13
2.2 Parallella.....13
2.3 Epiphany.....14
2.4 Epiphany III.....14
2.5 Parallella-16.....15

3 PRIMEROS PASOS.....19
3.1 Tarjeta microSD.....19
3.2 ESDK.....21
3.3 Manuales.....22
3.4 Ejemplos.....23
3.5 Buscar ayuda.....23

4 ARQUITECTURA.....25
4.1 Núcleos.....25
4.2 Malla 2D.....25
4.3 Memoria.....26
4.4 Modelos de programación.....29

5 HELLO WORLD.....31
5.1 ¿Qué hace este ejemplo?.....31
5.2 ¿Cómo vamos a estudiar este ejemplo?.....32
5.3 Iniciar el sistema (en el host).....32
5.4 Reserva de la memoria compartida (en el host).....33
5.5 Establecimiento de un grupo de trabajo (en el host).....33
5.6 Carga de los ejecutables y ejecución (en el host).....34
5.7 Reserva de la memoria compartida (en un eCore).....34
5.8 Escritura en la memoria compartida (en un eCore).....35
5.9 Lectura de la memoria compartida (en el host).....35
5.10 Visualización por pantalla (en el host).....35
5.11 Tareas de finalización (en el host).....36
5.12 Utilizar este ejemplo como plantilla.....36

6 ECLIPSE.....37
6.1 Abandono de Eclipse como IDE.....37
6.2 Incompatibilidad con ARM.....37
6.3 Esquema de desarrollo.....37
6.4 Instalar el eSDK.....38
6.5 Hello World con Eclipse.....39

7 DESARROLLO DEL PROYECTO.....45
7.1 Fases.....45
7.2 Entorno de trabajo.....46
7.3 Accediendo hasta Parallella-16.....47
7.4 Programando en Parallella-16.....48
7.5 Software utilizado.....49

8 EPIPHANY HOST LIBRARY (EHAL).....51
8.1 Introducción.....51
8.2 Funciones de Configuración de la Plataforma.....55
8.3 Funciones para Grupos de Trabajo y Memoria Externa.....59
8.4 Funciones de Transferencia de Datos.....64
8.5 Funciones de Control del System.....67
8.6 Funciones de Carga de Programas.....74
8.7 Funciones de Utilidad.....78

9 EPIPHANY HARDWARE UTILITY LIBRARY (ELIB).....85
9.1 Introducción.....85
9.2 Funciones de acceso al sistema de registros.....88
9.3 Funciones del Servicio de Interrupciones.....93
9.4 Funciones Timer.....99
9.5 Funciones de Movimiento de Datos y DMA.....105
9.6 Funciones para Mutex y Barrera.....116
9.7 Core ID and Workgroup Functions.....123

10 CONCLUSIONES.....133
10.1 Experiencia personal.....133
10.2 Líneas de investigación futuras.....134
10.3 Mi opinión sobre el proyecto Parallella.....135

11 ÍNDICE DE ILUSTRACIONES.....137

12 ÍNDICE DE TABLAS.....139

13 REFERENCIAS.....141

ANEXO: CONTENIDO DEL CD.....143
13.1 Memoria.....143
13.2 ESDK.....143
13.3 Ejemplos.....143

1 INTRODUCCIÓN

Desde hace unos años, podemos encontrar en el mercado ordenadores de bajo coste y de dimensiones reducidas. Uno de los más populares es Raspberry Pi, cuyo objetivo original era estimular la enseñanza de ciencias de la computación en las escuelas, y cuya versatilidad ha permitido que haya gente que lo utilice para crear desde centros de entretenimiento multimedia a máquinas recreativas caseras.

En este contexto, en el que ya no es una novedad encontrar ordenadores de este tipo, intenta hacerse un hueco Parallella-16, un ordenador cuya especificación y aspecto recuerda mucho a Raspberry Pi, pero que cuenta con algo que lo diferencia de otros: Lleva incorporado un coprocesador con 16 núcleos.

Vivimos en la era de los ordenadores con varios núcleos, donde la potencia de los ordenadores parece estar relacionada únicamente con esa variable, y en la que alguien podría pensar equivocadamente que si ejecutamos la misma aplicación en un Raspberry Pi y en un Parallella-16, los 16 núcleos extra de esta última harán que la aplicación se ejecute más rápido. Nada más lejos de la realidad. Para que una aplicación haga uso de esos 16 núcleos hay que diseñarla expresamente para ella, utilizando las herramientas y librerías que ofrece gratuitamente Adapteva, la empresa responsable de este proyecto.

Cualquiera con conocimientos de Linux y el lenguaje de programación C podría pensar que este es un inconveniente menor, pero la realidad es que sin la documentación adecuada, como es este caso, uno puede acabar invirtiendo más tiempo en aprender a utilizar esas herramientas y librerías, que en la complejidad propia de la aplicación a desarrollar. Y lo sé por experiencia.

He dedicado en exclusiva varios meses a estudiar cómo utilizar este ordenador, y he podido comprobar cómo muchas de las dudas que me han surgido, y que he podido leer en el foro oficial del proyecto, están relacionadas con el hecho de no tener una documentación actualizada, didáctica y acompañada de ejemplos lo suficientemente sencillos como para que la persona que lo vea no se pierda entre tanto código.

Este trabajo fin de grado intenta aportar, y es el contenido de los siguientes capítulos, una introducción acerca del proyecto y las características de este sistema, una revisión detallada del ejemplo "*Hola mundo*" que proporciona Adapteva, y la traducción de la documentación oficial acerca de sus librerías, a la que he aportado correcciones, notas personales y ejemplos didácticos.

Por tanto, este trabajo fin de grado tiene como objetivo facilitar las cosas a aquellas personas que quieran utilizar Parallella-16, un ordenador de bajo coste y de dimensiones reducidas que quiere acercar la supercomputación a todo el mundo.

2 PARALLELLA-16

Este capítulo es una introducción a Parallella-16. En él se habla desde la empresa responsable hasta su especificación técnica. Por otra parte, existe cierta confusión a la hora de utilizar varios nombres muy parecidos y relacionados con Parallella-16. Esta introducción permite también conocer claramente a qué hace referencia cada uno de ellos.

2.1 Adapteva

Adapteva es la empresa responsable de Parallella-16. Fue fundada en 2008 por Andreas Olofsson y tiene su base en Massachusetts. La compañía fue fundada con el objetivo de aumentar considerablemente la eficiencia energética de las operaciones en coma flotante de los dispositivos móviles. Su nombre es la combinación de "*adap*" y la palabra hebrea "*teva*" que significa naturaleza. El nombre es un reflejo de la filosofía de la empresa respecto a su forma de hacer negocios y la tecnología que quiere desarrollar.

2.2 Parallella

Parallella es el nombre del proyecto creado por Adapteva en KickStarter, una página para recaudar dinero para proyectos creativos como películas, videojuegos, música, etc. El objetivo era recaudar $ 750.000. El 28 de octubre de 2012 finalizó el plazo para recibir donaciones, obteniendo un total de $ 898.921. Este es el texto que acompañaba el proyecto.

Parallella: A Supercomputer For Everyone

Making parallel computing easy to use has been described as "a problem as hard as any that computer science has faced". With such a big challenge ahead, we need to make sure that every programmer has access to cheap and open parallel hardware and development tools. Inspired by great hardware communities like Raspberry Pi and Arduino, we see a critical need for a truly open, high-performance computing platform that will close the knowledge gap in parallel programing. The goal of the Parallella project is to democratize access to parallel computing. If we can pull this off, who knows what kind of breakthrough applications could arise? Maybe some of them will even change the world in some small but positive way.

2.3 Epiphany

La ilustración 2-1 muestra la evolución a lo largo de los años del desempeño de los ordenadores. En Adapteva creen que el futuro de la computación estará relacionado con la programación paralela y heterogénea, y pronostica que la transición a estos nuevos enfoques de programación se vería frenado alrededor de 2020 debido a un problema derivado de la arquitectura tradicional utilizada en el diseño de los ordenadores. Epiphany es la arquitectura propuesta por Adapteva para solventar este problema teniendo en cuenta la programación paralela y heterogénea.

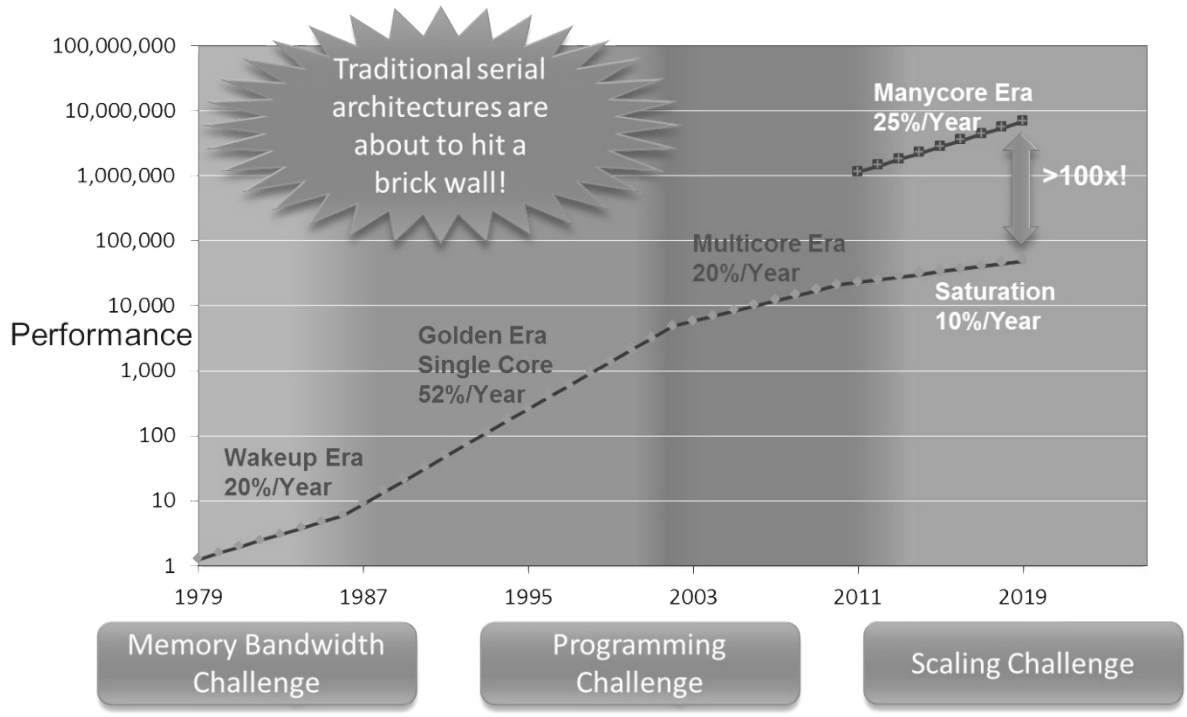


Ilustración 2-1. Evolución prevista de la arquitectura de ordenadores

2.4 Epiphany III

Epiphany III es un procesador diseñado para la arquitectura Epiphany. Al pertenecer a la 3ª generación de la arquitectura Epiphany, lo han etiquetado de esta forma.

Cuenta con 16 núcleos con procesadores RISC a 1GHz, los cuales soportan operaciones en coma flotante de 32 bits y tienen 512 GB/s de ancho de banda con la memoria local, que es de 32 KB (512 KB en total). Tiene 32 canales DMA independientes (2 por cada núcleo) y es programable en C/C++ y OpenCL.

Este procesador se ha diseñado desde cero teniendo en cuenta el FLOPS/Watt (número operaciones en coma flotante por segundo por vatio) en lugar del tradicional MIPS/Watt (millones de instrucciones por segundo por vatio). Esto lo hace ideal para dispositivos móviles, ya que da el mismo rendimiento que una GPGPU (*General Purpose Graphics Processing Unit*) con un consumo mucho menor.

2.5 Parallella-16

Y finalmente Parallella-16, el ordenador desarrollado por Adapteva gracias al proyecto Parallella. Es del tamaño de una tarjeta de crédito y solo consume 5W de potencia. Cuenta con un procesador Zynq-7000 y un coprocesador Epiphany III con 16 núcleos, de ahí su nombre.

Como decían en su proyecto para captar fondos, han intentado que este sea un proyecto realmente abierto, y se puede encontrar todos los detalles sobre esta plataforma en su página oficial.

Todos los que apoyaron el proyecto donando dinero han recibido una Parallella-16. Aquellos que quieran adquirir una pueden elegir entre 3 modelos:

Tabla 2-1. Especificaciones técnicas de Parallella-16

	P1600	**P1601**	**P1602**
Alias	"Microserver"	"Desktop"	"Embedded"
Epiphany III	E16G301	E16G301	E16G301
Núcleos	16	16	16
Memoria	1 GB DDR3	1 GB DDR3	1 GB DDR3
Voltaje	5V DC	5V DC	5V DC
Almacenamiento	Micro SD	Micro SD	Micro SD
Ethernet	10/100/1000	10/100/1000	10/100/1000
Xilinx Zynq	XC7Z010	XC7Z010	XC7Z020
USB	No	USB 2.0 Host Port	USB 2.0 Host Port
HDMI	No	Micro HDMI	Micro HDMI
Pins GPIO	0	24	48
Conectores eLink	0	2	2
FPGA	28K Logic Cells 80 DSP Slices	28K Logic Cells 80 DSP Slices	80K Logic Cells 220 DSP Slices
Precio (11/09/2014)	$ 119.00 / 146.46 €	$ 149.00 / 109.66 €	$ 249.00 / 183.26 €

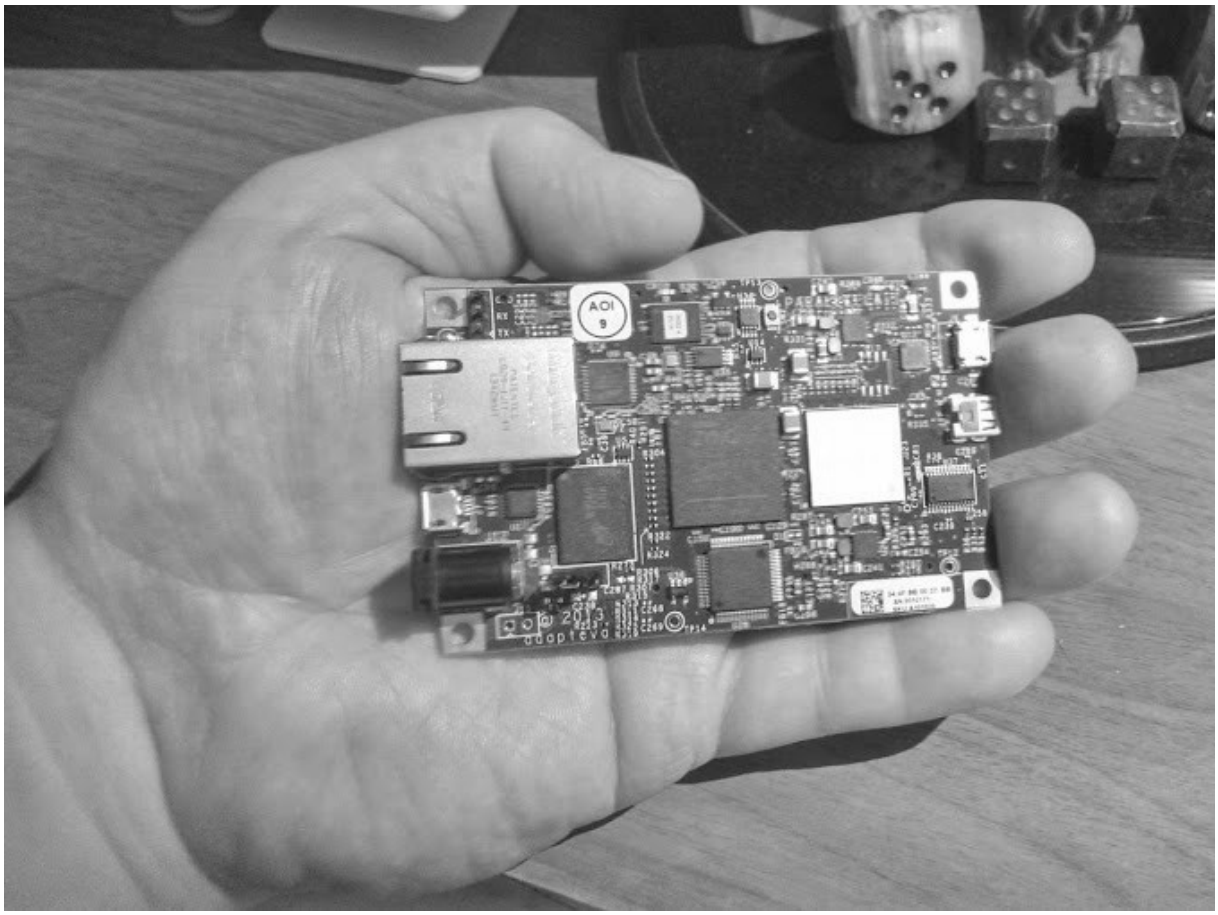


Ilustración 2-2. Parallella-16 sobre la palma de una mano

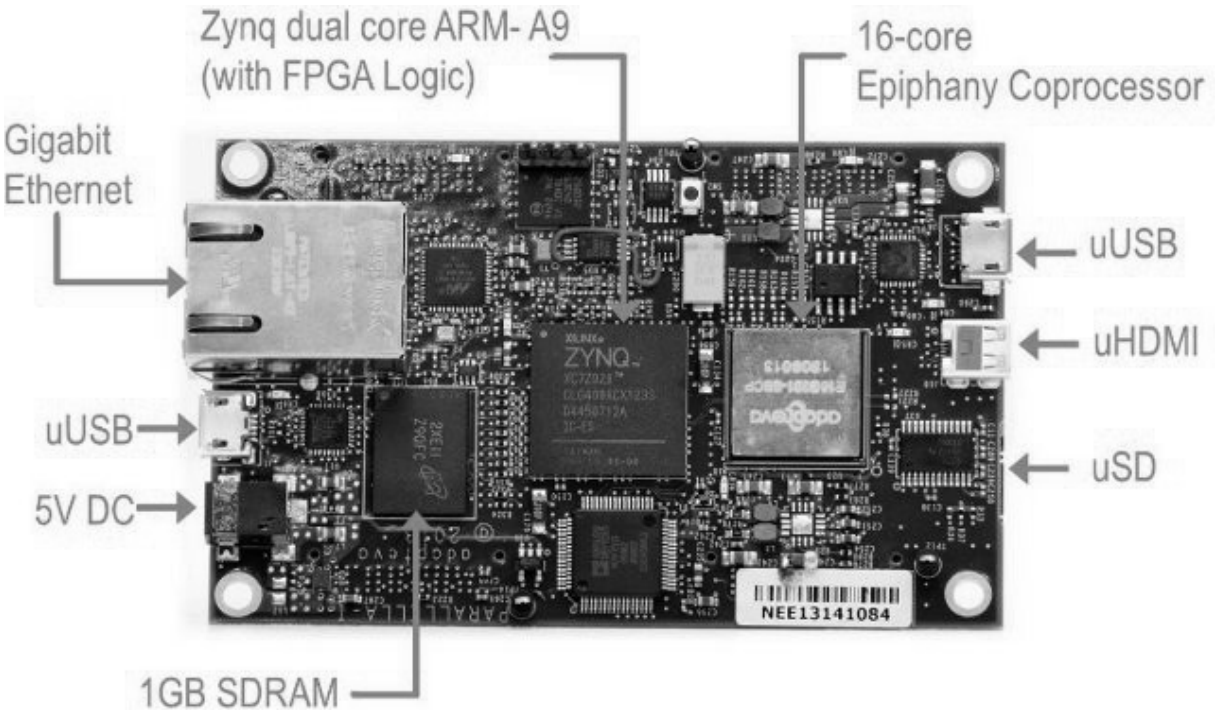


Ilustración 2-3. Componentes de Parallella-16

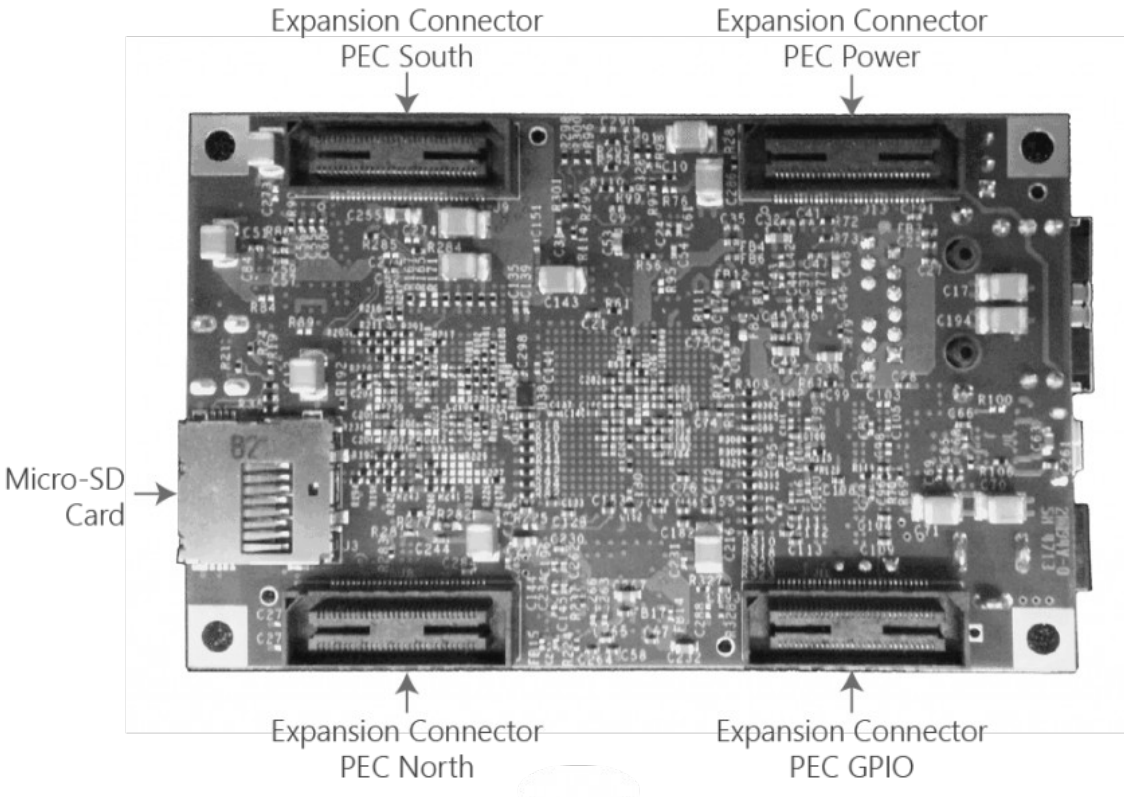


Ilustración 2-4. Parte inferior de Parallella-16

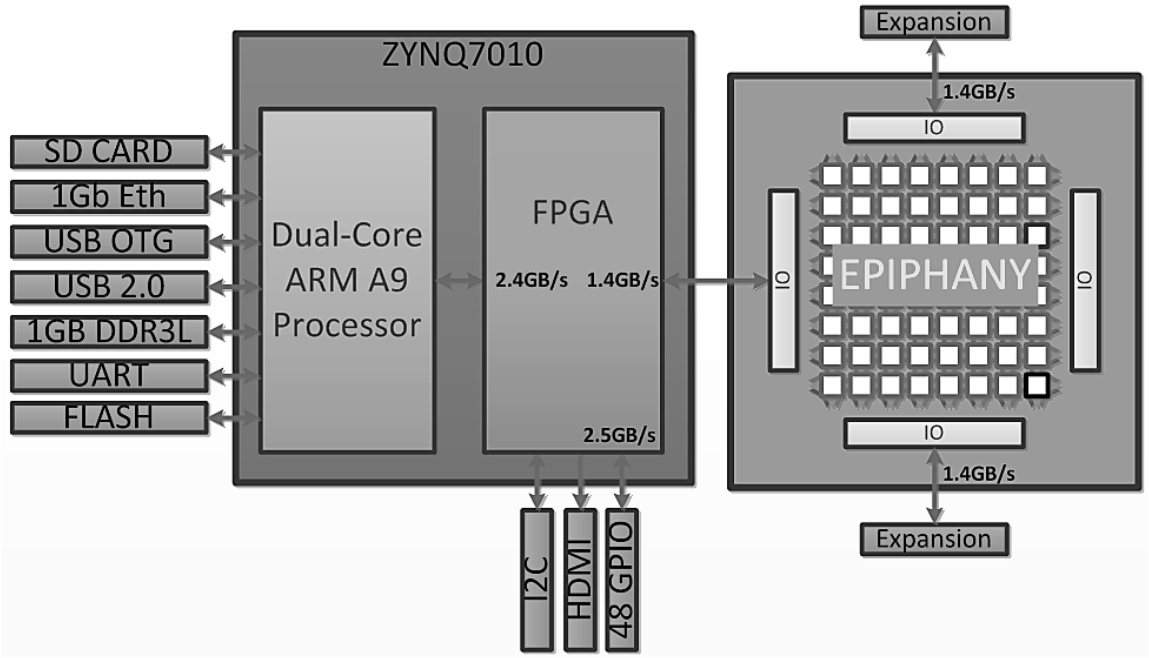


Ilustración 2-5. Conexión entre componentes de Parallella-16

3 PRIMEROS PASOS

Antes de empezar con los detalles técnicos, vamos a ver en este capítulo los primeros pasos que debemos dar con una Parallella-16, el software que necesitamos, los manuales y ejemplos disponibles, y muy importante, dónde acudir si necesitamos ayudas.

3.1 Tarjeta microSD

Parallella-16 utiliza una tarjeta microSD como disco duro, pero esta no viene incluida. Por tanto, uno de los primeros pasos es obtener una tarjeta microSD y añadir en ella el sistema operativo junto a las librerías necesarias. La página web oficial hay una sección dedicada a ello: *https://www.parallella.org/create-sdcard*.

Por tanto, vamos a esta página y descargamos la versión de Linux que Adapteva ha preparado para esta plataforma. En la actualidad la distribución utilizada es Ubuntu.

Hay que descargar además 2 archivos:

- Núcleo de Linux para Parallella con o sin soporte para HDMI.
- Archivos para la FPGA de Parallella.

Para seleccionar bien qué versión de estos archivos se debe descargar solo hay que tener claro cuántos núcleos tiene (16 o 64), el modelo del procesador Zynq y si tiene HDMI o no.

Una vez descargado todo, el siguiente paso es instalarlo en la tarjeta. Los pasos en **Windows** son:

1. Insertar la tarjeta en un ordenador con Windows.
2. Descomprimir la imagen de Ubuntu. Adapteva recomienda utilizar *7zip*.
3. Grabar la imagen de Ubuntu en la tarjeta microSD como haríamos con un CD. Adapteva recomienda utilizar *Win32 Disk Imager*.
4. Copiar los archivos `uImage` y `devicetree.dtb` (están comprimidos en el archivo que hemos descargado con el núcleo de Linux) y el archivo "`parallella-*.bin`".
5. Renombrar el fichero "`parallella-*.bin`" como "`parallella.bit.bin`".

IMPORTANTE: Por alguna razón que desconocemos, cuando seguimos estos pasos en el laboratorio la microSD no arrancaba. En cambio sí lo hizo cuando hicimos los pasos en Linux. Si intenta crear una microSD y no le funciona le recomiendo que intente seguir los pasos para Linux.

Los pasos en **Linux** son:

1. Insertar la tarjeta en un ordenador con Linux.
2. Descomprimir la imagen de Ubuntu.

   ```
   $ gunzip -d <releasename>.img.gz
   ```

3. Verifique la ruta de la tarjeta microSD.

   ```
   $ sudo fdisk -l | grep Disk
   ```

   Como el siguiente paso sobrescribirá los datos del disco seleccionado. Asegúrese de que la ruta es correcta. En el siguiente ejemplo la tarjeta microSD tiene la ruta "`/dev/mmcblk0/`". El resultado debe ser similar al siguiente.

   ```
   Disk /dev/sda: 500.1 GB, 500107862016 bytes

   Disk identifier: 0xa26fb586

   Disk /dev/mmcblk0: 15.9 GB, 15931539456 bytes

   Disk identifier: 0x000a07b6
   ```

4. Grabe la imagen de Ubuntu en la tarjeta microSD.

   ```
   $ sudo dd bs=64k if=<release-name>.img of=<sd-path>
   ```

   Deberá tener paciencia, ya que dependiendo del ordenador este proceso puede llegar a tardar más de 30 minutos.

5. Copie el núcleo de Linux para Parallella y los archivos del FPGA en la tarjeta microSD.

   ```
   $ tar -zxvf <kernel-*>.tgz -C <sd-path>/BOOT

   $ cp <parallella-*>.bin <sd-path>/BOOT/parallella.bit.bin

   $ sync
   ```

   Compruebe que puede ver las particiones 'rootfs' y 'BOOT' en la tarjeta microSD. Dependiendo de su configuración, usted puede o no necesitarlo para montar la partición. En Ubuntu, la ruta debería sería:

   ```
   "/media/$USER/BOOT"
   ```

Los pasos en **Mac** son:

1. Insertar la tarjeta en un ordenador con Mac:
2. Descomprimir la imagen de Ubuntu.
3. Verifique la ruta de la tarjeta microSD.

   ```
   $ diskutil list
   ```

4. Desmontar la tarjeta microSD.

   ```
   $ diskutil unmountDisk <sd-path>
   ```

5. Copia la imagen de Ubuntu en la tarjeta microSD.

   ```
   $ sudo dd if=<release-name>.img of=<sd-path> bs=1m
   ```

   Deberá tener paciencia, puede llevar bastante tiempo.

6. Copie el núcleo de Linux para Parallella y los archivos del FPGA en la tarjeta microSD.

   ```
   $ cp <parallella-*> <sd-path>/BOOT/parallella.bit.bin
   $ cp uImage <sd-path>/BOOT
   $ cp devicetree.dtb <sd-path>/BOOT
   ```

3.2 ESDK

En la documentación oficial, se denomina eSDK al SDK (el Kit de Desarrollo de Software) de Epiphany. Este kit está compuesto por:

- Compilador optimizado ANSI-C basado en *gcc*.
- Depurador multi-núcleo basado en *gdb*.
- Entorno de desarrollo Eclipse (solo para x86-64).
- Simulador funcional.
- Librerías para la comunicación entre núcleos.
- Manuales.
- Ejemplos.

Este kit viene instalado en la imagen de Ubuntu utilizada en la tarjeta microSD. En este documento hay un capítulo dedicado al entorno de desarrollo Eclipse, pero hay que advertir que este ya no tiene soporte de Adapteva, ya no se incluye en las nuevas versiones del eSDK. En el foro oficial algunos usuarios han optado por utilizar *Code::Block* como entorno alternativo.

3.3 Manuales

Adapteva pone a disposición de todo el mundo en su página web oficial 4 manuales y una serie de documentos que van desde presentaciones a informes, algunos realizados por entidades como Australia National University, Computer Journal o Ericsson.

- **Parallella Reference Manual (Septiembre 2014)** – Es un manual en el que podemos encontrar los detalles de Parallella-16, desde un mapa de la placa hasta los detalles eléctricos.
- **Epiphany III (E16G301) Datasheet (Febrero 2013)** – Es una ficha de datos que ha acabado siendo un manual. En él se pueden encontrar los datos más relevantes acerca del procesador Epiphany III, el modelo con 16 núcleos.
- **Epiphany-IV (E64G401) Datasheet (Junio 2013)** – Es la ficha de datos del procesador Epiphany-IV, el modelo con 64 núcleos.
- **Parallella Schematic (Diciembre 2013)** – Son los planos de la circuitería eléctrica de Parallella-16.
- **Epiphany Architecture Reference Manual (Marzo 2014)** – Aquí podemos encontrar todo lo referente a aspectos técnicos relacionados con la plataforma Epiphany. En el siguiente capítulo, dedicado a su arquitectura, puede encontrar un resumen de este manual.
- **Epiphany SDK Reference Manual (Septiembre 2013)** – Este manual habla del eSDK y en él podemos encontrar desde cómo compilar hasta las funciones de las librerías para el manejo de los núcleos.

Uno de los problemas de utilizar Parallella-16 es son sus manuales. La fecha que acompaña al título de los manuales es la fecha de la versión que actualmente (14/01/2015) están colgados en su página web.

Un ejemplo del estado de estos manuales lo podemos ver en *Parallella Reference Manual*. La página del capítulo 6, dedicado al arranque de Parallella-16 está, salvo por el título, en blanco.

Otro ejemplo, que puede hacer que una persona invierta mucho tiempo en buscar dónde está el error en el código de *hello-world*, es que si nos vamos al documento *Epiphany SDK Reference Manual* podemos encontrar la descripción de la función *e_reset_core()*, la función utilizada en el ejemplo *hello-world*, cuando esta función ya no se encuentra en el eSDK y ha sido sustituida por *e_reset_group()*.

3.4 Ejemplos

Aunque el eSDK viene con algunos ejemplos, Adapteva tiene en GitHub un repositorio de ejemplos entre los que cabe destacar:

- **john** – Una versión para Parallella del programa "*John the Ripper*" utilizado para descifrar contraseñas por fuerza bruta.
- **kinect_test** – Una demostración de Kinect que utiliza Epiphany para colorear, escalar y renderizar.

El problema de los ejemplos que aporta Adapteva es que no son didácticos. El ejemplo *hello-world* que viene con el eSDK no utiliza los 16 núcleos a la vez, sino uno por uno, y el siguiente ejemplo, una multiplicación de matrices, es tan extenso que es difícil sacar de él algo en claro.

En el capítulo dedicado a las librerías de programación, a medida que se describen las funciones se ofrecen varios ejemplos que se pueden encontrar en el CD que acompaña a este trabajo fin de grado. El propósito de estos ejemplos es mostrar cómo utilizar la función, no utilizarla en una aplicación útil. De manera que una persona con conocimientos de C pueda comprender para qué sirve y cómo utilizarla, por ejemplo, una barrera.

3.5 Buscar ayuda

La forma ideal de buscar ayuda es entrar en el foro (*https://parallella.org/forums*) que Adapteva ha creado para interactuar con la gente. Dentro del foro, existen varias categorías dedicadas en exclusiva a hardware, software, proyectos, etc. Cada una de ellas divididas en subcategorías.

En Freenode, una red de servidores IRC, se puede encontrar un canal #parallella en la que se pueden resolver problemas en tiempo real, pero la presencia de usuarios no siempre es síntoma de presencia de personas. Por lo que aconsejo utilizar mejor el foro, donde quizás alguien haya buscado solución al mismo problema.

4 ARQUITECTURA

En este capítulo se abordan temas relacionados con la arquitectura Parallella-16. No pretende sustituir al manual oficial sobre su arquitectura, sino simplemente un resumen de lo más relevante antes de ponerse a programar en ella.

4.1 Núcleos

Cada uno de los núcleos es de tipo RISC a 1 GHz. Cuenta con una memoria local de 32 kB que a su vez forma parte de la memoria externa (compartida) del sistema. Una FPU (*Floating Point Unit*) permite operaciones en coma flotante de 32 bits con precisión simple. Estas operaciones son: Suma, resta, multiplicación más suma, multiplicación más resta, conversiones entre fijo y flotante, valor absoluto. Por cada ciclo de reloj es capaz de realizar 1 operación de enteros o 2 en coma flotante.

4.2 Malla 2D

En la arquitectura Epiphany los núcleos de sus procesadores forman una malla 2D, de forma que, los núcleos de Parallella-16 una malla de 4x4. La arquitectura permite conectar múltiples Epiphany III, de manera que conectamos nuestra Parallella-16 a otras 3, la malla sería de 64x64[1].

Cada núcleo tiene una interface de red que utiliza para comunicarse con un router que lo conecta a la malla. Como puede observarse en la ilustración 4-1, estos routers, uno por núcleo, conectan entre sí a todos los núcleos. Su latencia baja les permite comunicarse entre sí a una velocidad muy alta, 512 GB/s de ancho de banda con la memoria local según la documentación.

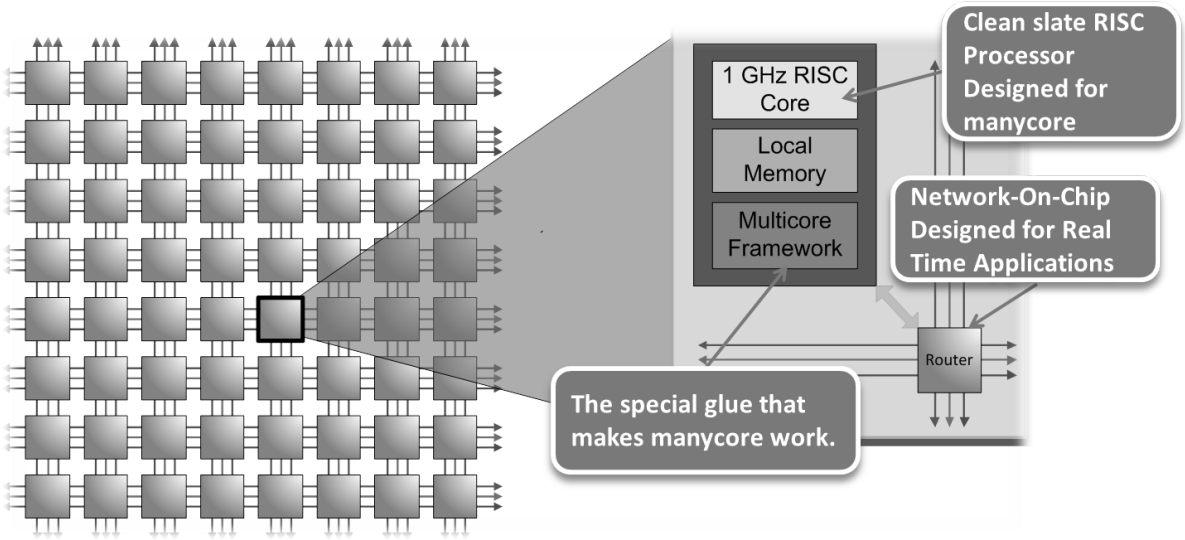


Ilustración 4-6. Arquitectura multi-núcleo Epiphany

[1] La documentación no deja claro si el número de chips conectados debe ser potencia de 2, es decir, no dice si es posible conectarla a otra o si es necesario al menos tres.

4.3 Memoria

La arquitectura Epiphany utiliza 32 bits para direccionar su memoria. En las imágenes se utilizan 8 dígitos hexadecimales para representar direcciones, y es la notación utilizada en los ejemplos. Esta notación es más fácil y rápida de escribir que 32 bits, y permite ver una relación directa entre dirección y núcleo que se explica más abajo.

Cada uno de los núcleos cuenta con 32 kB de memoria local repartido en 4 bancos de memoria, cada uno de 8 kB. Los núcleos son capaces de realizar 1 operación carga de memoria de 64 bits por cada ciclo de reloj.

Como la arquitectura está basada en un mapa de memoria externa (compartida), esta memoria local de los núcleos es accesible tanto por el procesador principal ARM como por cada uno de los núcleos del chip Epiphany III. Por tanto, el memoria asociada al chip es 512 kB (32 kB x 16 núcleos).

Para el flujo de datos cada núcleo cuenta con 2 canales DMA (Acceso Directo a Memoria), con lo cual, el chip tiene 32 canales DMA independientes para el acceso a la memoria externa (compartida).

Como puede verse en la ilustración 4-2, los núcleos están distribuidos a lo largo de la memoria. Los primeros 3 dígitos hexadecimales (los más altos) identifican a qué núcleo pertenece una dirección de memoria y se utiliza como ID del núcleo.

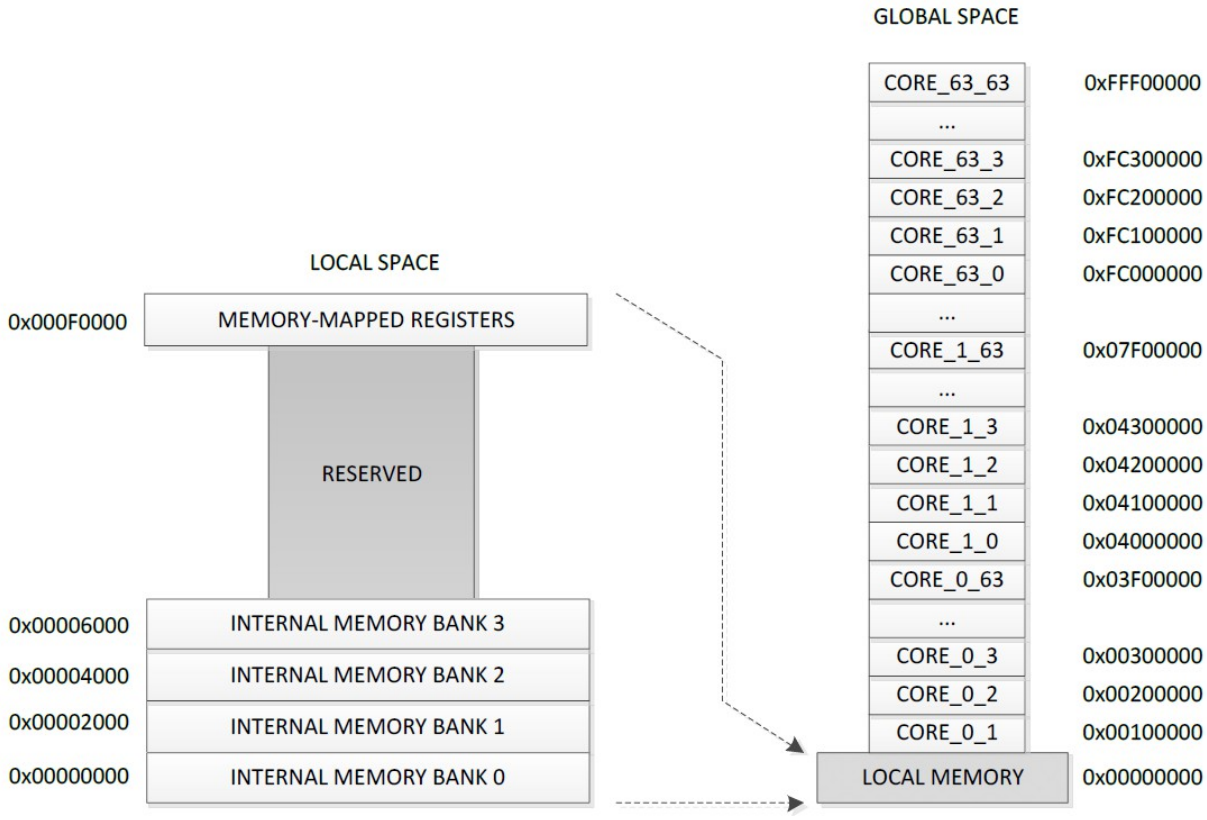


Ilustración 4-7. Memoria de la arquitectura Epiphany

Si tomamos la dirección 0x0420100, esta pertenece al núcleo 042 (en hexadecimal). Para conocer las coordenadas de este núcleo hay que pasar esta cifra de hexadecimal a binario (000001000010) y descomponer la cifra en fila (000001) y columna (000010). Es decir, la dirección 0x0420100 pertenece al núcleo (1, 2).

Hay que tener en cuenta una cosa. En la ilustración 4-3 se muestran las coordenadas de los núcleos de cuatro Parallella-16 conectadas para trabajar como una sola. Como se puede observar, las coordenadas no son las que esperábamos. En el caso de la Parallella-16 que hemos utilizado en el laboratorio el primer núcleo es el (32, 8).

Este detalle de las coordenadas de los núcleos a nivel de arquitectura no es realmente necesario a la hora de programar. Desde el procesador ARM las coordenadas utilizadas son relativas al grupo de núcleos definidos por la aplicación, estas si empiezan en (0, 0), y desde los núcleos, aunque sí podemos ver estos valores, realmente no son necesarios conocerlos, ya que las funciones permiten programar de forma que podemos ignorarlas, se consultan simplemente para pasarlas a otras funciones.

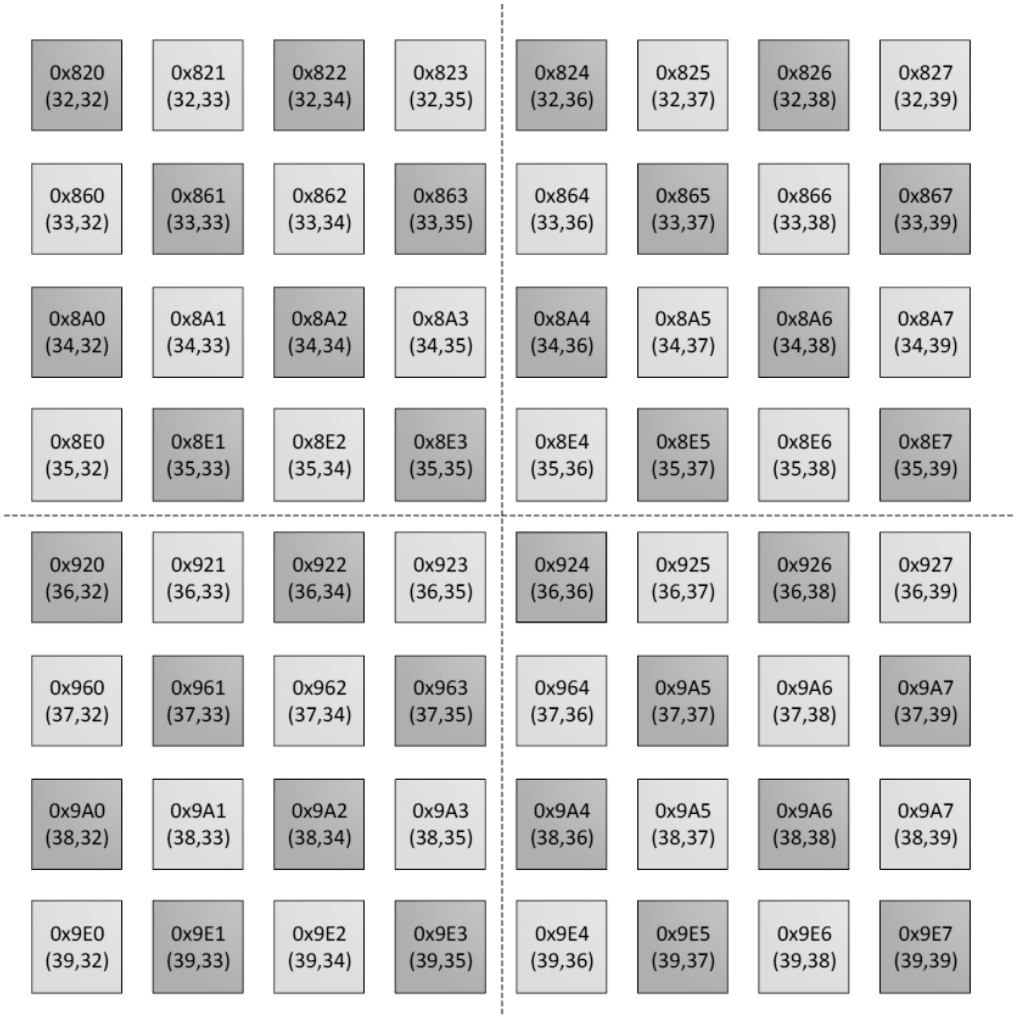


Ilustración 4-8. Malla 2D de cuatro Parallellas-16

Un detalle que no viene en el manual sobre la arquitectura de Epiphany, pero sin embargo sí aparece en el relacionado con el SDK, incluyendo un ejemplo para dejarlo bien claro, es el hecho de que la memoria externa (compartida), vista desde el procesador ARM puede tener una dirección y vista desde los núcleos puede tener otra, ya que acceden de forma distinta, el host con el bus del sistema y los núcleos por eLinks. Por eso, aunque tengan cada uno un puntero a la misma zona de memoria, sus direcciones pueden ser diferentes al estar mapeadas.

Parallella-16 está configurada por defecto con 32 MB de memoria DRAM (RAM Dinámica), que vista desde el host va desde `0x1e000000` a `0x1fffffff`, y para superar ciertas limitaciones se ha mapeado estas direcciones en los núcleos que ve la memoria desde la dirección `0x8e00000` hasta la `0x8fffffff`.

Las direcciones de base (real y alias) de la memoria externa (compartida) se encuentran en el archivo HDF (*Hardware Description File*). En la carpeta `/bsps` del eSDK podemos encontrar los archivos HDF correspondientes a Parallella-16 y cada uno de sus prototipos. Si lo abrimos podemos consultar estos y otros detalles.

```
// Platform description for the
// Parallella/1GB/E16G3
PLATFORM_VERSION   PARALLELLA1601
ESYS_REGS_BASE         0x808f0f00

NUM_CHIPS                       1
CHIP                      E16G301
CHIP_ROW                       32
CHIP_COL                        8

NUM_EXT_MEMS                    1
EMEM                    ext-DRAM
EMEM_BASE_ADDRESS      0x3e000000
EMEM_EPI_BASE          0x8e000000
EMEM_SIZE              0x02000000
EMEM_TYPE                    RDWR
```

La configuración realizada para Parallella-16 crea una variable de entorno para que al programar se tome por defecto un archivo HDF concreto. Hay que asegurarse de que el archivo HDF pertenece al modelo que estamos utilizando. Podemos consultar el valor de esta variable de entorno escribiendo lo siguiente:

```
Linaro:~> echo $EPIPHANY_HDF
/opt/adapteva/esdk/bsps/current/plataform.hdf
```

A la hora de programar, el núcleo no necesita utilizar una dirección como 0x8e00000 para crear una variable en la memoria externa (compartida). En los ejemplos se puede ver cómo declarar una variable en la memoria externa (compartida) sin necesidad de conocer su dirección. Pero es importante conocer esta diferencia si se pretende programar a un nivel más bajo.

Por último, y con relación a las direcciones que apuntan al mismo sitio pero son distintas en vistas desde el procesador ARM y Epiphany III, está el hecho de que los núcleos pueden declarar punteros a su memoria local (bancos de memoria) utilizando una dirección relativa al núcleo. Es decir, el banco de memoria 1 está situado en la dirección 0xXXX2000, siendo XXX el ID del núcleo. Pues bien, un puntero que apunte a la dirección 0x2000 estará direccionando a su banco de memoria 1. Esto permite crear fácilmente un puntero al banco 1, y utilizando una función, hacer que ese puntero apunte al banco 1 de otro núcleo.

Todos estos usos y detalles pueden verse en varios ejemplos, como los que hacen referencia al uso de funciones relacionadas con los núcleos vecinos y el uso de un buffer en la memoria externa (compartida).

4.4 Modelos de programación

A la hora de programar, la comunicación entre el *host* (procesador ARM) y los núcleos (procesador Epiphany) se realiza de alguna de estas dos maneras:

- **Buffer compartido**. Tanto el *host* como los núcleos acceden a una misma zona de memoria. Es la técnica utilizada en el ejemplo *hello-world* del siguiente capítulo.
- **Bancos de memoria**. Los núcleos cuentan cada uno con 4 bancos de memoria. El *host* puede acceder y escribir en ellos los datos que quiere pasar al núcleo, o puede acceder y leer de ellos los datos que el núcleo haya dejado para el host.

Por otra parte, según la documentación oficial, la arquitectura Epiphany cuenta con un modelo de programación neutral y compatible con los métodos de programación paralela más populares, entre los que se incluye:

- **SIMD (Single Instruction Multiple Data)** – Los núcleos recibirían el mismo programa y distintos datos, y su ejecución estaría sincronizada, de manera que todos ellos ejecutarían a la vez la misma instrucción. Los datos sobre los que operaría serían distintos. Este modelo puede ser implementado utilizando el mismo programa en todos los núcleos y utilizando las barreras como método de sincronización. El propósito de una barrera es detener la ejecución del programa hasta que todos los núcleos lleguen a esa misma instrucción.

- **SPMD (Single Program Multiple Data)** – Es una subcategoría de SIMD. Aquí los núcleos recibirían el mismo programa y distintos datos, pero su ejecución no estaría sincronizada, de manera que cada núcleo puede ejecutar sus instrucciones independientemente de que el resto haya llegado o no a la misma instrucción. Este modelo equivale a utilizar el mismo programa en todos los núcleos sin tomar ningún método de sincronización.

- **MIMD (Multiple Instruction Multiple Data)** – En este caso cada núcleo recibiría un programa distinto y datos distintos. Este modelo equivale a utilizar un programa distinto en cada núcleo.

- **Systolic Array** - Aquí los datos de salida de un núcleo sería la entrada de otro, de manera que solo dos núcleos mantendrían comunicación con el procesador principal (ARM), el primero que recibe los datos y el último que los devuelve ya manipulados. Este modelo se implementaría utilizando la función que permite conocer cuál es el siguiente núcleo vecino y utilizando la capacidad de escribir y leer sobre la memoria local de cada núcleo.

- **Shared-Memory Multithreading** – Los núcleos utilizarían una misma variable compartida. Para utilizar este modelo los núcleos deben declarar la misma variable en la memoria externa (compartida) y utilizar un mutex para evitar los problemas de acceso concurrente a la misma.

- **MPI (Message Passing Interface)** – En el repositorio que Adapteva tiene en GitHub (*https://github.com/parallella*) existe una versión de *hello-world* que utiliza MPI para realizar este ejemplo utilizando el pase de mensajes. El mismo ejemplo ofrece una versión utilizando OpenCL y OpenMP. Es tal la importancia de MPI que en el foro oficial tiene una sección aparte, pero nada indica que sea Adapteva el responsable de la librería utilizada.

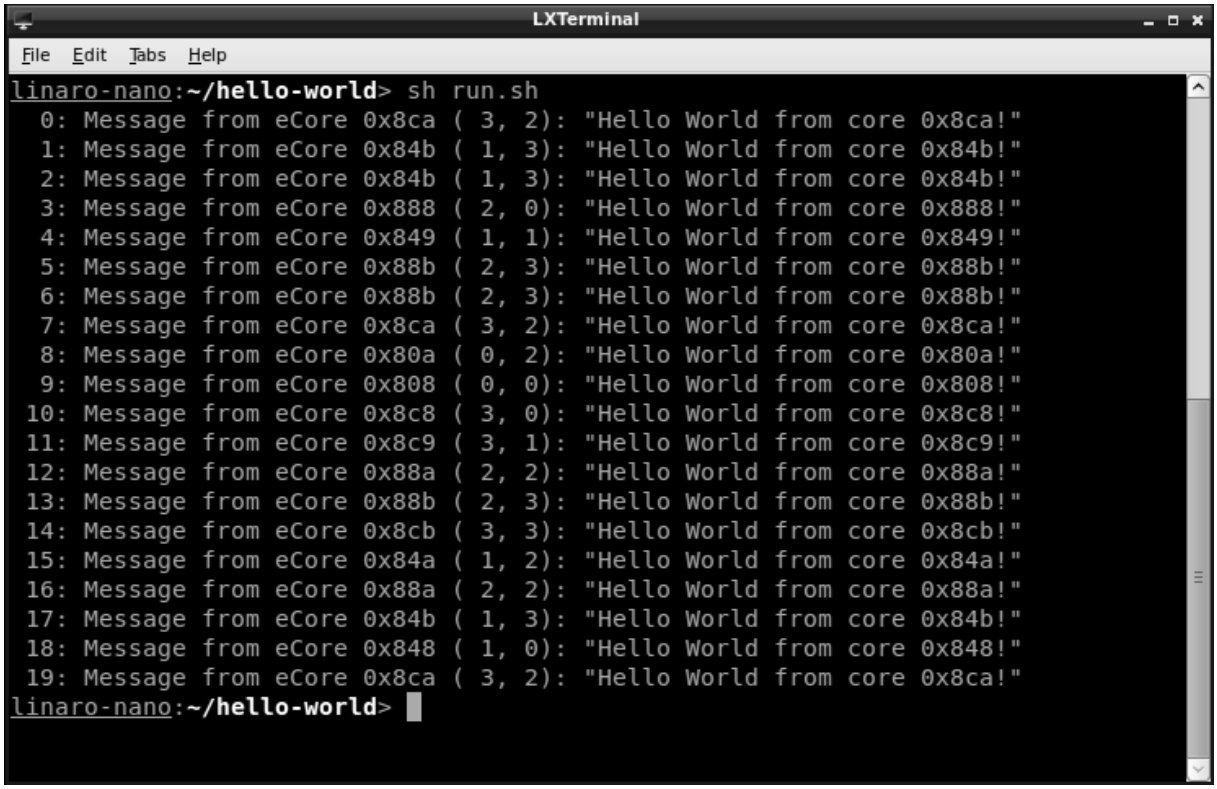
5 HELLO WORLD

Este capítulo es una introducción a la programación en Parallella-16. Para ello se utiliza el ejemplo *hello-world* que viene con el eSDK y podemos encontrarlo en la siguiente ruta: */opt/adapteva/esdk/examples/hello-world*. En él podemos ver los elementos comunes a todos los programas realizados en esta arquitectura.

IMPORTANTE: Debido a cambios en la API, se recomienda primero leer este capítulo antes de probar el ejemplo.

5.1 ¿Qué hace este ejemplo?

Para ver lo que hace este ejemplo podemos ir a su carpeta en la carpeta de ejemplo, compilarlo y ejecutarlo utilizando los archivos *build.sh* y *run.sh*. Lo que veremos serán los típicos mensajes de un ejemplo "*hello world*", pero de varios núcleos.



Hay que decir que, en realidad, este ejemplo no hace uso de varios núcleos a la vez, sino que los usa uno a uno. El ejecutable principal (en el procesador ARM) utiliza un bucle, y en cada iteración utiliza un núcleo distinto de Epiphany III.

Otra cosa a destacar es que, podríamos pensar que son los núcleos los que están escribiendo en pantalla, pero no es así. Los núcleos no pueden imprimir en pantalla, solo el procesador principal. El núcleo almacena la cadena "hello world..." en la memoria externa (compartida). El ejecutable principal lee esta cadena desde la memoria externa (compartida) y es quien la muestra en pantalla.

5.2 ¿Cómo vamos a estudiar este ejemplo?

Para simplificar el análisis de este ejemplo vamos a dividirlo en fases, y vamos a ver qué funciones están implicadas en cada fase. Estas fases, y las funciones utilizadas en este ejemplo, son prácticamente las mismas que vamos a poder encontrar en cualquier aplicación diseñada para esta arquitectura.

Este ejemplo consta de dos ejecutables distintos, uno que ejecuta el procesador ARM y el que ejecutan los núcleos. El código fuente de ambos ejecutables vamos a dividirlo en las siguientes fases para su estudio por separado:

#	Ejecutable principal (en el host)	Ejecutable secundario (en un eCore)
1	Inicialización del sistema.	
2	Reserva de la memoria externa.	
3	Establecimiento de un grupo de trabajo	
4	Carga de los ejecutables y ejecución.	
5		Reserva de la memoria externa.
6		Escritura en la memoria externa.
7	Lectura de la memoria externa.	
8	Visualización por pantalla.	
9	Tareas de finalización	

NOTA: En gris las fases que se repiten en cada bucle.

5.3 Iniciar el sistema (en el host)

En el siguiente código, la primera función se encarga de establecer la conexión con Epiphany III. En el capítulo dedicado a la librería *e-hal.h* se explicará por qué se pasa NULL. La segunda línea se encarga de poner a punto el sistema. Y finalmente, la tercera línea obtiene información sobre el procesador Epiphany, de esta forma se puede conocer el número de núcleos que tiene y las coordenadas del primero.

```
// initialize system, read platform params from
// default HDF. Then, reset the platform and
// get the actual system parameters.
e_init(NULL);
e_reset_system();
e_get_platform_info(&platform);
```

5.4 Reserva de la memoria compartida (en el host)

En el siguiente código, la función *e_alloc()* sirve para reservar memoria dentro de la memoria externa (compartida). La variable *emem* no se utilizará como buffer para comunicarse con el procesador principal (host), sino que apunta a una parte de la memoria externa.

En la declaración de las variables podemos ver que el tamaño es de 128 (bytes) y el desplazamiento sobre la memoria (offset) tiene un valor de en hexadecimal de 0x01000000.

¿Por qué ese desplazamiento? El sistema reserva por defecto 32 MB para código y datos, siendo los primeros 16 MB para código y los siguientes 16 MB para datos. Un salto de 0x01000000 (16777216 = 16 x 2^{20}) equivale a saltarse esos 16 MB. Cuando el núcleo cree el buffer en la memoria externa (compartida), el buffer estará dentro de la memoria apuntada por *emem*.

```
// Allocate a buffer in shared external memory
// for message passing from eCore to host.
e_alloc(&emem, _BufOffset, _BufSize);
```

5.5 Establecimiento de un grupo de trabajo (en el host)

Antes de cargar un ejecutable en los núcleos de Epiphany III hay que crear un grupo de trabajo, es decir, indicar qué núcleos vamos a utilizar. En el siguiente código, la primera función sirve para establecer un grupo de trabajo.

En el capítulo dedicado a la arquitectura del sistema vimos que sus núcleos forman una matriz 4x4. Un grupo de trabajo es un subconjunto de esa matriz y se establece indicando las coordenadas del primer núcleo, en el ejemplo (*row, col*), y el número de filas y columnas que tendrá, en este caso, el grupo de trabajo es tan solo de 1x1.

La siguiente función sirve para poner a punto el núcleo (0, 0) dentro de ese grupo de trabajo.

```
// Open the single-core workgroup and reset the core, in
// case a previous process is running. Note that we used
// core coordinates relative to the workgroup.
e_open(&dev, row, col, 1, 1);
e_reset_core(&dev, 0, 0);
```

IMPORTANTE: Con la actualización de la API, esta función ya no existe. En su lugar se utiliza la función **e_reset_group()**, que pone a punto todos los núcleos del grupo de trabajo, de esa forma no es necesario ir uno a uno.

5.6 Carga de los ejecutables y ejecución (en el host)

En el siguiente código, la primera función carga el ejecutable "*e_hello_world.srec*" en el núcleo que ocupa la posición (0, 0) dentro del grupo de trabajo *dev*. Otra función permite cargar el mismo ejecutable en todos los núcleos del grupo de trabajo.

El parámetro *E_TRUE* le indica al núcleo que en cuanto tenga el ejecutable lo ejecute. En el capítulo dedicado a la librería *e-hal.h* veremos que en algunas ocasiones necesitaremos retardar el inicio de esa ejecución, utilizando las funciones *e_start()* y *e_start_group()* Para indicar el comienzo de la ejecución.

Como vemos, tras cargar el ejecutable ejecuta la función *usleep()* para detener la ejecución del programa 10.000 microsegundos (10 milisegundos), permitiendo al núcleo finalizar su programa. La lectura que hace a continuación la veremos más tarde.

```
// Load the device program onto the selected eCore
// and launch after loading.
e_load("e_hello_world.srec", &dev, 0, 0, E_TRUE);

// Wait for core program execution to finish, then
// read message from shared buffer.
usleep(10000);
e_read(&emem, 0, 0, 0x0, emsg, _BufSize);
```

5.7 Reserva de la memoria compartida (en un eCore)

En la documentación oficial hace referencia a los núcleos de Epiphany como eCore. Ahora que el núcleo ha cargado el ejecutable en un núcleo y ha detenido su ejecución, vamos a ver qué ocurre en el núcleo.

Para comunicarse con el *host*, el procesador principal, el núcleo va a crear un array de caracteres, que utilizará como buffer, en la memoria externa (compartida). Este array está situado dentro de la memoria apuntada por la variable *emem* del *host*, al principio de la memoria externa (compartida).

El núcleo, para situar el buffer al inicio de la memoria externa (compartida) utiliza la palabra reservada *SECTION* indicando dónde quiere situarlo. En el capítulo 5 del manual del SDK aparece una lista de secciones definidas: "*data_bank1*", "*code_dram*", etc.

```
char outbuf[128] SECTION("shared_dram");
```

5.8 Escritura en la memoria compartida (en un eCore)

Para pasarle los datos al host el núcleo solo debe escribir en el buffer. Para hacerlo, utiliza la función *sprintf()* de la librería *stdio.h*, que permite escribir sobre cadena los datos utilizando un determinado formato

```
// The PRINTF family of functions do not fit
// in the internal memory, so we link against
// the FAST.LDF linker script, where these
// functions are placed in external memory.
sprintf(outbuf, "Hello World from core 0x%03x!", coreid);
```

5.9 Lectura de la memoria compartida (en el host)

Una vez finalizada la ejecución del núcleo, el host lee la memoria externa (compartida) utilizando la variable *emem*, que apuntaba al inicio de esta.

La función e_read() permite leer la memoria externa (compartida) o la memoria local de un núcleo. En este caso, como el primer parámetro es *emem*, se lee de la memoria externa (compartida). Los siguientes parámetros representan las coordenadas (0, 0), pero como no se va a leer la memoria local de un núcleo, estos dos parámetros son ignorados. El siguiente parámetro 0x0 indica el desplazamiento (offset) sobre la memoria. En este caso no hay desplazamiento, se lee al principio de la memoria externa (compartida). Por último se le pasa un array de caracteres y la cantidad de bytes a leer, que coincide con la cantidad de caracteres (1 carácter ocupa 1 byte en esta arquitectura).

```
// Wait for core program execution to finish, then
// read message from shared buffer.
usleep(10000);
e_read(&emem, 0, 0, 0x0, emsg, _BufSize);
```

5.10 Visualización por pantalla (en el host)

Como ya se ha dicho, los núcleos no pueden imprimir por pantalla. En este caso, el host imprime por pantalla el mensaje recogido en la memoria externa (compartida). Por alguna razón, el autor del código ha preferido utilizar la función *fprintf()*, pero no porque no se pueda utilizar *printf()*.

```
// Print the message and close the workgroup.
fprintf(stderr, "\"%s\"\n", emsg);
e_close(&dev);
```

5.11 Tareas de finalización (en el host)

Como ya se ha dicho anteriormente, al inicio del bucle este programa crea un grupo de trabajo con un solo núcleo. Al final del bucle cierra o elimina el grupo de trabajo para crear otro nuevo. Para ello utiliza *e_close()*.

Cuando el bucle finaliza, antes de finalizar la ejecución del programa, se realizan dos operaciones. La primera es liberar la memoria reservada utilizando *e_free()*, y después utilizar *e_finalize()* para finalizar la conexión con Epiphany III.

```
      // Print the message and close the workgroup.
      fprintf(stderr, "\"%s\"\n", emsg);
      e_close(&dev);
}
// Release the allocated buffer and finalize the
// e-platform connection.
e_free(&emem);
e_finalize();

return 0;
```

5.12 Utilizar este ejemplo como plantilla

A la hora de crear nuestras propias aplicaciones, el ejemplo *hello-world* puede utilizarse como plantilla. Si cambiamos el nombre de los archivos, o si añadimos más, por ejemplo porque vayamos a utilizar diferentes ejecutables en diferentes núcleos, habrá que realizar los cambios apropiados en los archivos build.sh y run.sh.

En el archivo *build.sh* podemos distinguir 3 líneas:

- La primera línea que utiliza los archivos fuentes utiliza *gcc* para compilar el archivo que ejecutará el procesador ARM. Si su archivo cambia el nombre habrá que modificar esta línea. El ejecutable tiene la extensión ELF.
- Otra línea que utiliza los ficheros fuente utiliza *e-gcc*, el compilador basado en *gcc* para compilar los ejecutables para los núcleos.
- La línea que utiliza *e-objcopy* realmente no utiliza ningún archivo fuente del programa, pero utiliza el archivo compilado con *e-gcc*. Si se cambia la línea anterior probablemente haya que modificar también esta línea.

El archivo *run.sh* básicamente lanza la ejecución del archivo ELF del ARM. Si su nombre cambió habrá que modificar también este archivo.

6 ECLIPSE

Este capítulo es, en parte, una traducción del que se puede encontrar en el manual *Epiphany SDK Reference*. En él se muestra cómo configurar la versión de Eclipse que Adapteva proporciona para utilizarlo como entorno de desarrollo integrado para desarrollar programas que utilicen Parallella-16.

Aunque Adapteva ya no proporciona Eclipse en sus últimas actualizaciones del eSDK, este venía con la versión utilizada en este proyecto fin de grado, y parece lógico añadir este capítulo.

6.1 Abandono de Eclipse como IDE

Hasta llegar a Parallella-16, Adapteva ha tenido algunos prototipos como EMEK3 y EMEK4. Con estos prototipos, la versión de Eclipse ha funcionado correctamente.

Con la llegada de Parallella-16, el producto final, Adapteva advertía en su manual de referencia del eSDK que Eclipse estaba desactualizado y que tenía problemas con la sincronización con Parallella-16, y con el lanzamiento y depuración de aplicaciones.

Ya hay en el FTP del proyecto Parallella versiones del eSDK más actualizadas que la utilizada para este proyecto fin de grado, y en estas versiones ya no se puede encontrar este entorno de desarrollo.

6.2 Incompatibilidad con ARM

La versión modificada de Eclipse no funciona con procesadores ARM, como es el caso de Parallella-16. Está pensada para ejecutarse en un sistema Linux de 64 bits.

Hoy día, crear aplicaciones sin un entorno de desarrollo parece algo del pasado, pero existen pocos o ningún entorno de desarrollo a la altura de lo que cualquiera esperaría encontrar en una arquitectura como i386 o amd64. Por eso Adapteva ha apostado por ofrecer Eclipse, aunque no se ejecute en Parallella-16.

6.3 Esquema de desarrollo

El esquema de desarrollo con Eclipse y Parallella-16 es el siguiente. Si conectamos a una misma red un ordenador y Parallella-16, podemos crear y compilar las aplicaciones en el ordenador con Eclipse y ejecutarlas en Parallella-16.

El eSDK cuenta con una aplicación llamada *e-server* que arrancada en Parallella-16 permite ejecutar aplicaciones a través de TCP/IP. Eclipse, al que previamente se le ha indicado la IP de Parallella-16, hace las funciones de cliente y lanza la aplicación, mostrando en su consola el resultado de dicha ejecución.

6.4 Instalar el eSDK

Para instalar el eSDK, que incluye Eclipse, en un ordenador aparte de Parallella-16 se recomienda utilizar la versión Ubuntu 12.4 de 64 bits. Esta es la versión más cercana en el tiempo a cuando se escribió el manual, y cuenta con paquetes necesarios que en versiones posteriores de Ubuntu han sido sustituidos por otros.

Para instalar el eSDK basta con seguir los pasos que vienen en el manual. Utilizar un terminal para ejecutar estas instrucciones:

1. Instalar los siguientes paquetes:

```
sudo apt-get install libgmp3-dev libexpat1-dev openjdk-6-jre
libmpfr-dev libmpc-dev tcsh csh  g++
```

2. Crear un directorio para el eSDK:

```
sudo mkdir -p /opt/adapteva/
```

3. Descargar el eSDK (* será un número que indica la versión) y descomprimir en la carpeta anterior:

```
sudo tar xzf esdk.*.linux_x86_64.tar.gz -C /opt/adapteva/
```

4. Crear un enlace simbólico al eSDK:

```
sudo ln -sTf /opt/adapteva/esdk.* /opt/adapteva/esdk
```

5. Copiar y pegar estas líneas:

```
echo 'EPIPHANY_HOME=/opt/adapteva/esdk' >> ${HOME}/.bashrc
echo '. ${EPIPHANY_HOME}/setup.sh' >> ${HOME}/.bashrc
```

Para comprobar si todo ha ido bien, hay que cerrar el terminal y volver a abrirlo, para que las modificaciones del archivo `.bashrc` tengan efecto. Una vez hecho esto, ejecutamos la instrucción `e-gcc –version`.

```
$ e-gcc --version
e-gcc (Epiphany toolchain (built 20130910)) 4.8.2 20130729
Copyright (C) 2013 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
```

6.5 Hello World con Eclipse

Si hemos seguido los pasos anteriores, Eclipse se encontrará en la carpeta `/opt/adapteva/esdk/tools/eclipse`. Es bueno comprobarlo, hay versiones del eSDK que pueden haberlo puesto en una carpeta distinta.

Es importante no hay que arrancarlo haciendo doble clic sobre él, hay que hacerlo a través del terminal, ya que de esa forma, tienen en cuenta los parámetros que se han introducido en el archivo `.bashrc`.

Podemos arrancar Eclipse ejecutando la siguiente línea:

```
${EPIPHANY_HOME}/tools/eclipse/eclipse &
```

Crear el proyecto

Para crear el ejemplo vamos a *File > New > C Project*. Como muestra la ilustración 6-1, escribimos el título de la aplicación, eligiendo como prototipo *Hello World C Project*, que está dentro de *Epiphany Executable (Multi Core)*, y pulsamos *Next*.

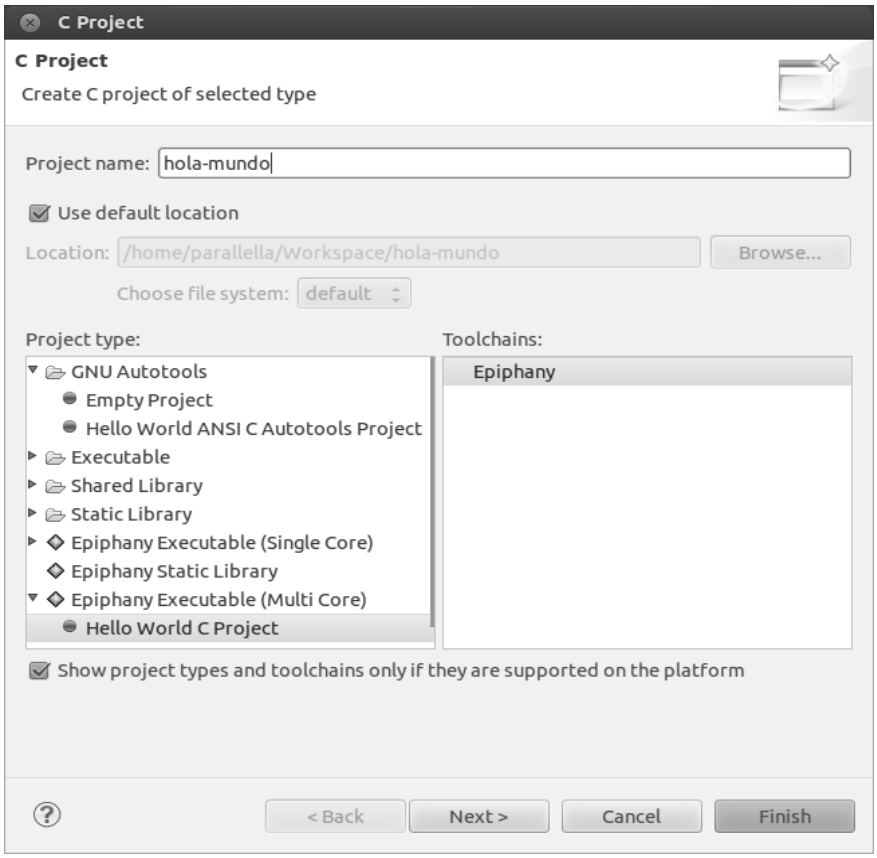


Ilustración 6-9. Creación de un proyecto en Eclipse

En la siguiente ventana podemos seleccionar cuantos núcleos queremos utilizar, y las coordenadas del primero. Como ya se dijo en el capítulo relacionado con la arquitectura, el núcleo tiene unas coordenadas que no son las que esperamos. Hay que comprobar cuáles tiene nuestra Parallella-16. En el caso del utilizado para este trabajo fin de grado es (32, 8).

En este ejemplo, como se muestra en la ilustración 6-2, se utilizarán 16 (4x4) indicando que las coordenadas del primer núcleo es (32, 8). Pulsar *Next*, y en la siguiente ventana *Finish*.

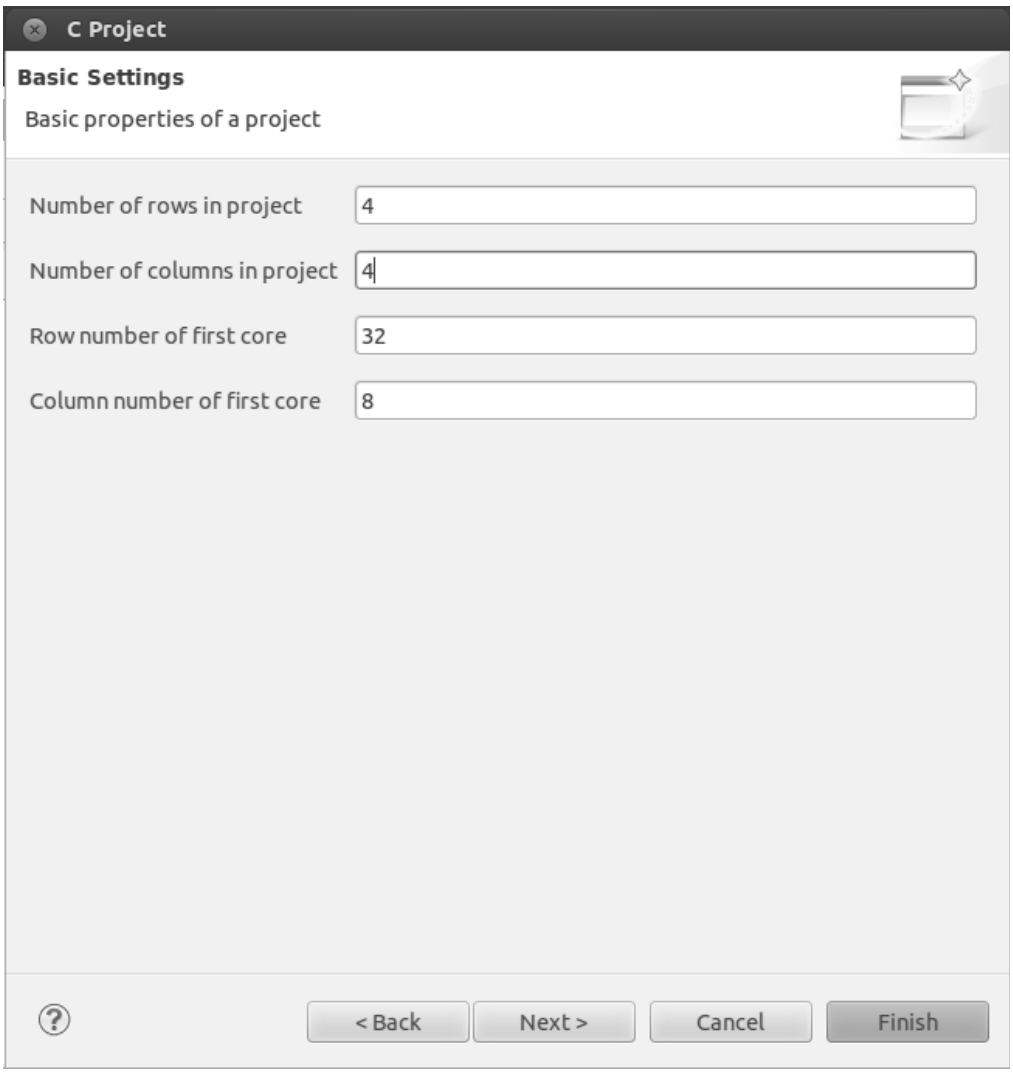


Ilustración 6-10. Indicando cuantos núcleos utilizar en Eclipse

El resultado es que Eclipse crea 16 proyectos (uno por cada núcleo), y otros dos. Uno con el nombre que hemos indicado, otro al que se le ha concatenado `_commonlib`, y los 16 restantes con el núcleo al que va dirigido (`.core.row_col`).

Configuración del proyecto

A continuación hay que hacer una serie de cambios, ya que viene configurado por defecto para utilizar los prototipos EMEK3 y EMEK4, y no para Parallella-16.

Haciendo clic con el botón secundario del ratón sobre el proyecto dirigido a un núcleo, hacemos clic en *Properties*. En la ventana desplegamos *C/C++ Builder* y hacemos clic en *Settings*. Hay que buscar *Linker Description File*, y veremos como el archivo utilizado por defecto es para EMEK3. La ilustración 6-3 muestra dónde se encuentra esta opción dentro de la ventana de propiedades del proyecto.

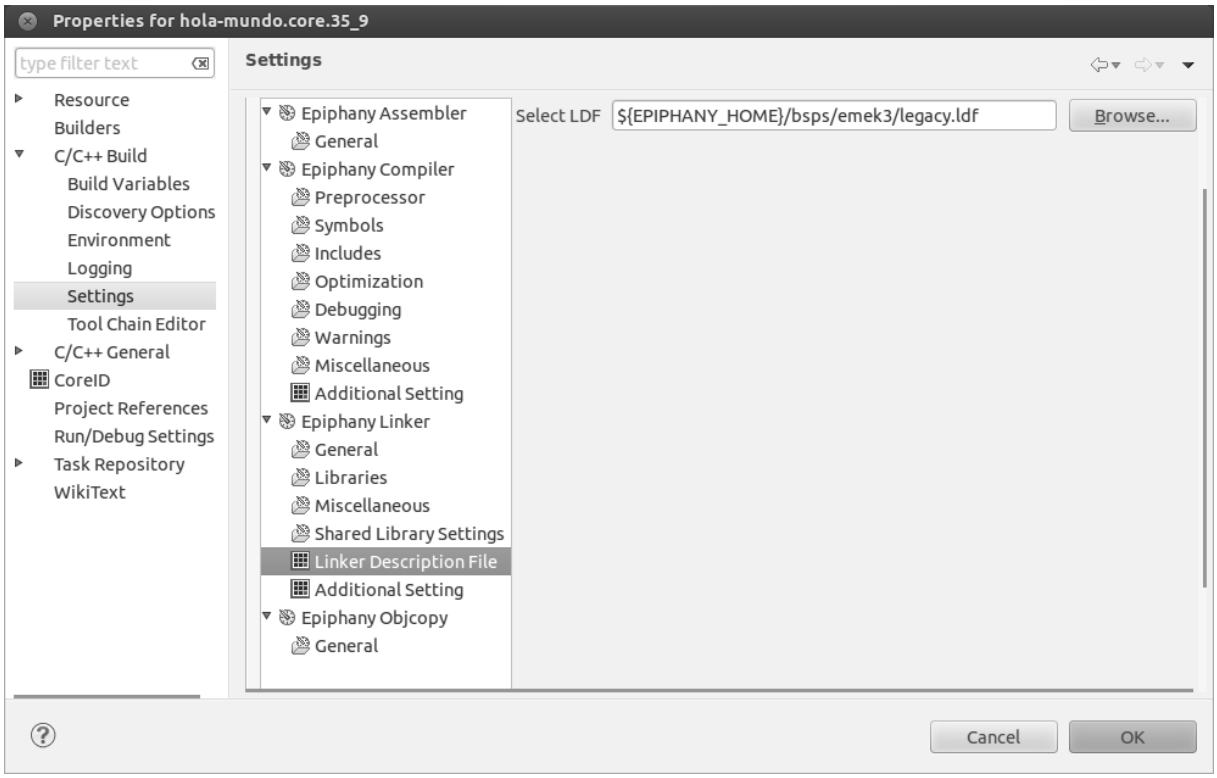


Ilustración 6-11. Propiedades del proyecto en Eclipse

Si se está utilizando, como en este trabajo fin de grado, Parallella-16, hay que indicarle el archivo adecuado. Dentro de la carpeta `/bsps` se puede encontrar carpetas que hacen referencia a otros prototipos y modelos. No es difícil encontrar el archivo *legacy.ldf* correcto.

Hay que hacer varios pasos más si queremos que la compilación del proyecto no de errores. El primero es incluir, desde esta misma ventana de propiedades en *Setting*, en *Libraries*, la librería "*e-lib*".

Ahora aplicamos estos cambios a todos los proyectos, haciendo clic con el botón secundario del ratón sobre el proyecto modificado, y seleccionando *Apply C/C++ setting for other core projects*. Seleccionamos el resto de proyectos.

Aplicación e-server

En Parallella-16 debemos ejecutar el comando e-server. Esta es una herramienta proporcionada por el eSDK que hace que Parallella-16 ejecute un servidor y quede a la escucha a través de los puertos que muestra al inicio de su ejecución.

La ilustración 6-4 muestra el resultado de la ejecución de este comando. Como puede observarse, el comando indica el archivo HDF utilizado para su ejecución, en este caso, al no haberle proporcionado ninguno utiliza el que viene configurado por defecto a través de la variable de entorno `EPIPHANY_HDF`.

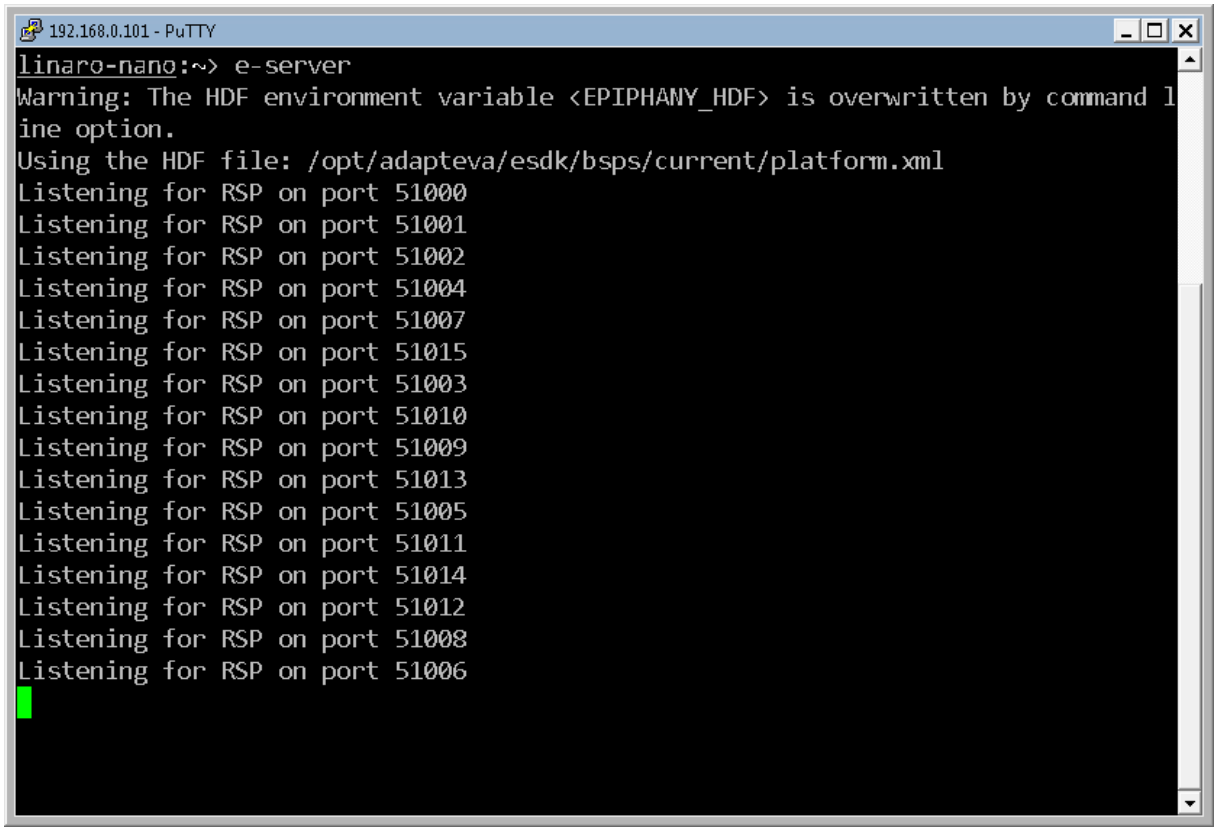


Ilustración 6-12. Ejecución del e-server

Compilar

Ya podemos compilar el proyecto sin errores, aunque dependiendo de la versión de Linux, puede que no encuentre ciertas librerías que, estando en el sistema, no tienen la versión esperada. Basta con ver en el error de qué librería se trata, buscar la librería en el sistema y crear un enlace simbólico con el nombre esperado por el compilador.

Depurar

Para lanzar y depurar la aplicación hay que ir al menú *Run* > *Debug Configurations…* y con el botón secundario del ratón hacer clic sobre *Epiphany Multicore Applications (gdb)* y seleccionar *New*. En la ventana que aparece hay que seleccionar *Enable auto build* en la pestaña *Projects* y hacer clic en el botón *Debug*.

Para conocer la IP de Parallella-16 podemos utilizar el comando `ifconfig`.

Eclipse compilará la aplicación y establecerá conexión con Parallella-16, algo que puede verse en el monitor de este último. Eclipse intentará cambiar la perspectiva del entorno al modo de depuración, como se puede ver en la ilustración 6-5. Para ver el saludo ejecutado en el núcleo hay que ir a la consola.

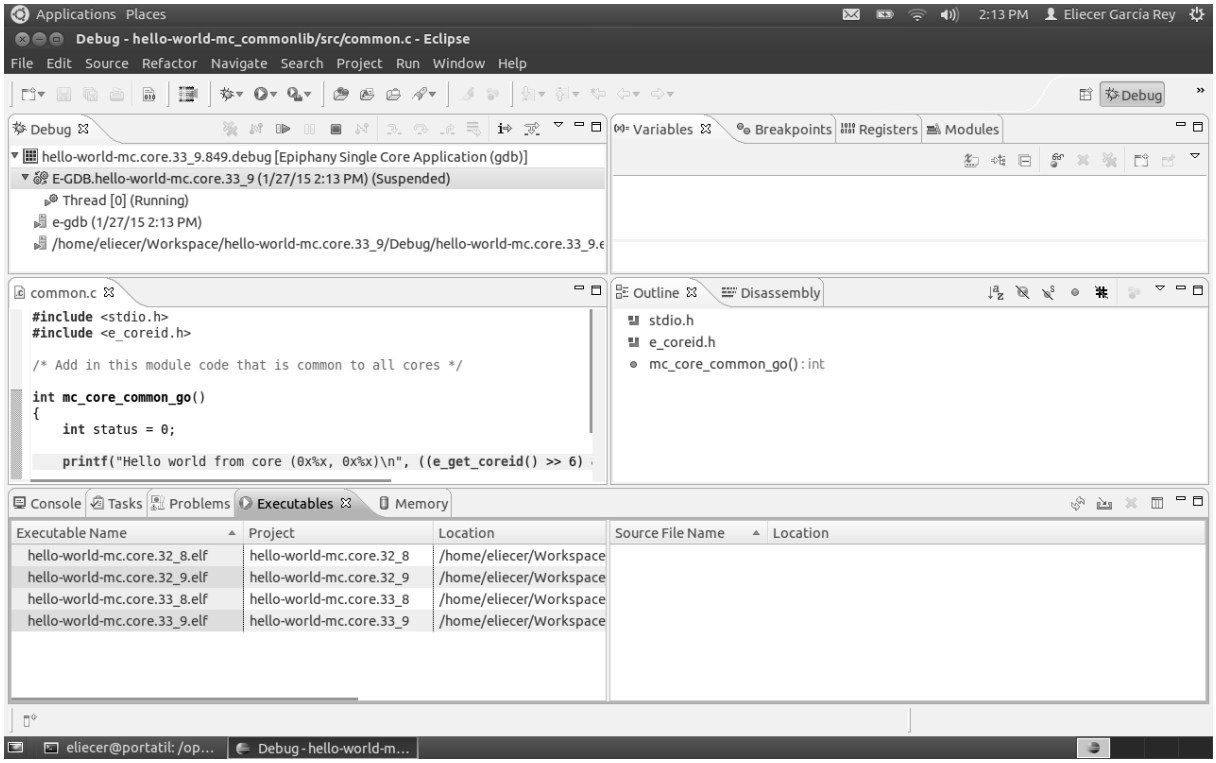


Ilustración 6-13. Eclipse en modo depuración ejecutando hello-world

Ejecutar

Para ejecutar la aplicación hay que ir al menú *Run* > *Run Configurations…* y con el botón secundario del ratón hacer clic sobre *Epiphany Multicore Applications (loader)* y seleccionar *New*.

7 DESARROLLO DEL PROYECTO

En este capítulo se detalla el desarrollo del proyecto, esto es, las distintas fases en las que se puede dividir el trabajo realizado desde el principio, hasta las herramientas utilizadas.

7.1 Fases

Las fases en las que se ha dividido este trabajo, visto a *posteriori*, han sido:

1. **Toma de contacto**. En esta fase he intentado documentarme. He buscado qué era Parallella-16, quién lo fabrica, si tiene página web, qué manuales hay al respecto, dónde se puede buscar ayuda, etc.

2. **Configuración del entorno**. Siguiendo lo leído en los manuales, en esta fase he intentado configurar el entorno de Parallella-16 y mi ordenador personal para poder trabajar con los ejemplos del eSDK.

3. **Ejemplo *Hello world***. En circunstancias normales la ejecución del ejemplo *hello-world* no constaría como una fase, pero por el tiempo invertido hasta echarlo a andar lo voy a considerar como una fase.

   Resulta que la versión del eSDK de Parallella-16 era distinta al que podía utilizar en mi ordenador, y por este motivo, el ejemplo *hello-world* de mi ordenador no se compilaba en Parallella-16. De una versión a otra del eSDK había funciones que habían dejado de existir y se habían cambiado por otras.

   Hasta hallar el problema, un cambio no documentado, pasó bastante tiempo, y decidí revisar una a una todas las funciones de la documentación. Es así como surgió la idea de traducir las librerías.

4. **Traducción**. En esta fase traducía la parte del manual del eSDK relacionado con las librerías. Esto me permitió conocer qué funciones podía utilizar, y con ello, qué mecanismos de comunicación y sincronización tenía.

5. **Ejemplos**. En esta fase desarrollé ejemplos que utilizaban casi por completo todas las funciones traducidas en la fase anterior. En el foro oficial he visto como mucha gente preguntaba cosas básicas sobre las funciones debido a que el manual dedica pocas líneas a documentar cada función. Una vez creados los ejemplos, con mi experiencia y las dudas que he visto en el foro he revisado la traducción hecha de las funciones y he completado con apreciaciones su descripción.

6. **Memoria y presentación**. Esta ha sido la última fase del proyecto. Una vez hecho los ejemplos y corregida la traducción, junto a toda la experiencia acumulada en este trabajo, he empezado a escribir la memoria, a la que he incorporado la traducción de las librerías, y la presentación.

7.2 Entorno de trabajo

Aunque Parallella-16 es un ordenador que puede trabajar de forma independiente, se trata de un recurso compartido con otras universidades, por lo que se ha habilitado el acceso remoto desde fuera de la facultad. En concreto, la Parallella-16 utilizada en este trabajo fin de grado se encuentra en el laboratorio 3.3.11 de la Escuela Superior de Ingeniería Informática de la Universidad de Málaga.

El acceso remoto planteó un problema, el apagado y el encendido de Parallella-16. Para solucionarlo se ha utilizado un dispositivo que funciona como un relé USB. Este dispositivo, conectado a uno de los ordenadores del laboratorio, controla la fuente de Parallella-16. De esta forma, accediendo a este ordenador, no solo se puede encender y apagar Parallella-16, sino que también se puede acceder a él mediante una conexión SSH o VNC.

La ilustración 7-1 muestra a la izquierda el ordenador del laboratorio a través del cual accedía a Parallella-16. A la derecha la fuente de alimentación y la Parallella-16 a la que se le ha puesto un disipador. Entre ambos ordenador se puede ver el relé con los cables de la fuente de alimentación. En la siguiente página se pueden ver imágenes de estos componentes por separado.

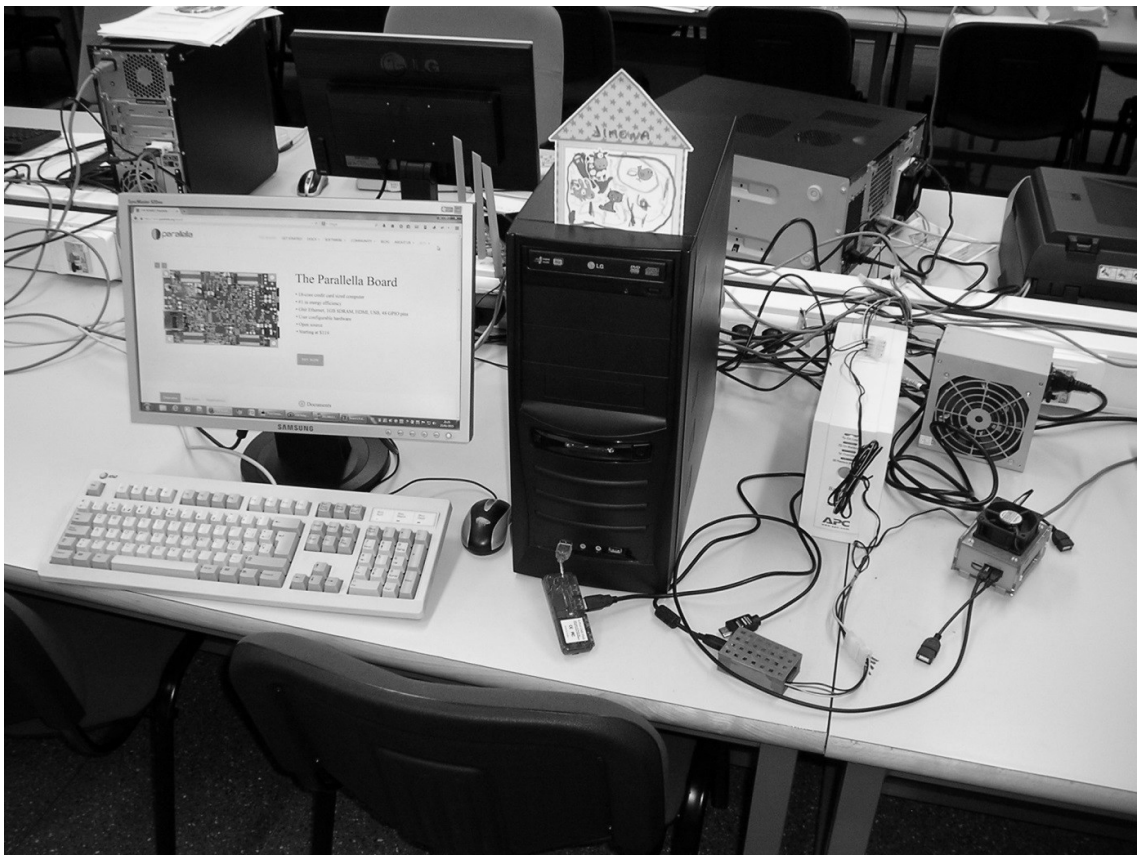


Ilustración 7-14. Configuración PC y Parallella-16

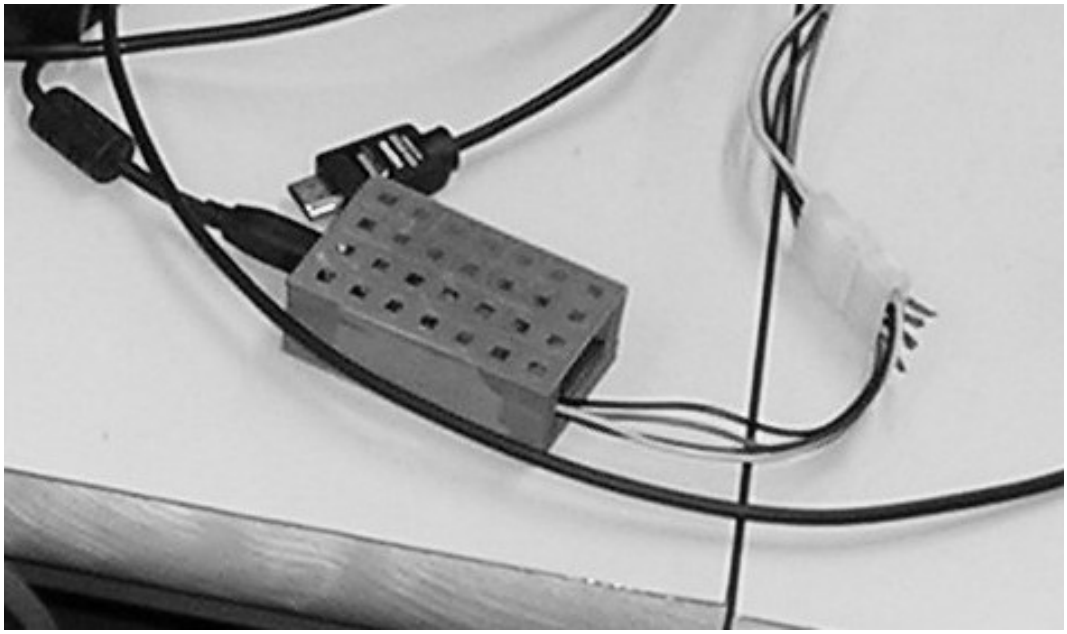


Ilustración 7-15. Relé USB con los cables de alimentación

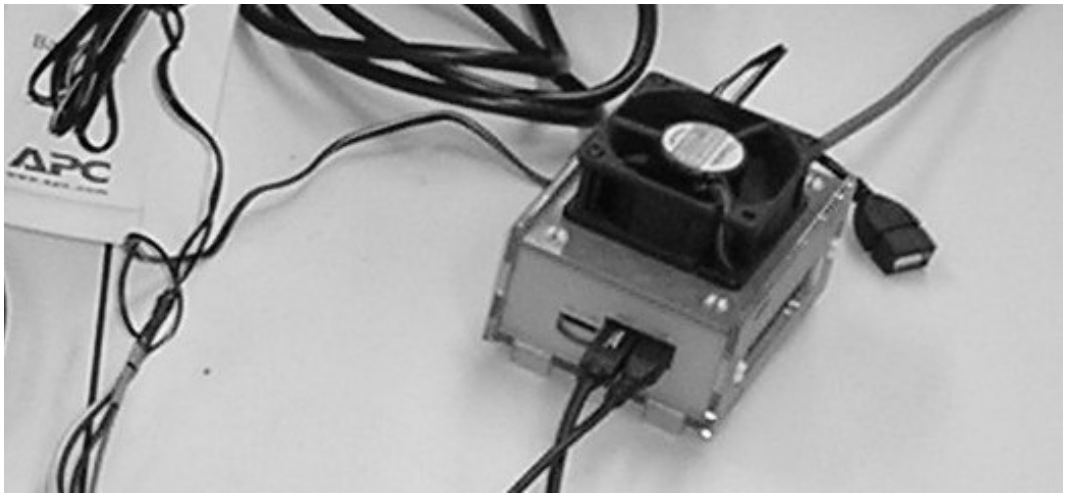


Ilustración 7-16. Parallella-16 con un disipador de calor

7.3 Accediendo hasta Parallella-16

La mayor parte del tiempo, el acceso a Parallella-16 ha sido mediante acceso remoto. Los pasos necesarios para acceder a él han sido:

1. Conectar mediante escritorio remoto con el ordenador del laboratorio.
2. Una vez que hemos accedido a este ordenador, se activa la fuente de alimentación de Parallella-16 utilizando el software que controla el relé.
3. Una vez arrancado Parallella-16, desde el mismo ordenador que hemos utilizado para arrancarlo, utilizamos PuTTY para conectar mediante SSH. Esto permite utilizar Parallella-16 desde un terminal en modo texto.

7.4 Programando en Parallella-16

Al principio, para programar accedía a Parallella-16 de forma visual y no en modo texto con PuTTY. Para ello arrancaba un servidor VNC en Parallella-16. A continuación, desde el ordenador del laboratorio que controlaba su fuente de alimentación, utilizando UltraVNC Viewer accedía de forma visual. Para cargar y descargar ficheros en Parallella-16 utilizaba WinSCP, también desde el ordenador del laboratorio.

Como Parallella-16 no contaba con ningún entorno de desarrollo ni editor de texto visual que resaltara la sintaxis, salvo Emacs, la única ventaja de conectar de forma visual era el uso del ratón y la navegación mediante ventanas.

Con el tiempo empecé a utilizar Emacs sin tener conocimientos por la única razón de que resaltaba la sintaxis. En cuanto aprendí a manejar ficheros, cortar y pegar, dividir la ventana y acceder al terminal sin salir de Emacs, pasé a utilizar Emacs en modo texto. Al final acabé accediendo a Parallellla-16 únicamente con PuTTY y trabajando con Emacs en modo texto, como muestra la ilustración 7-4.

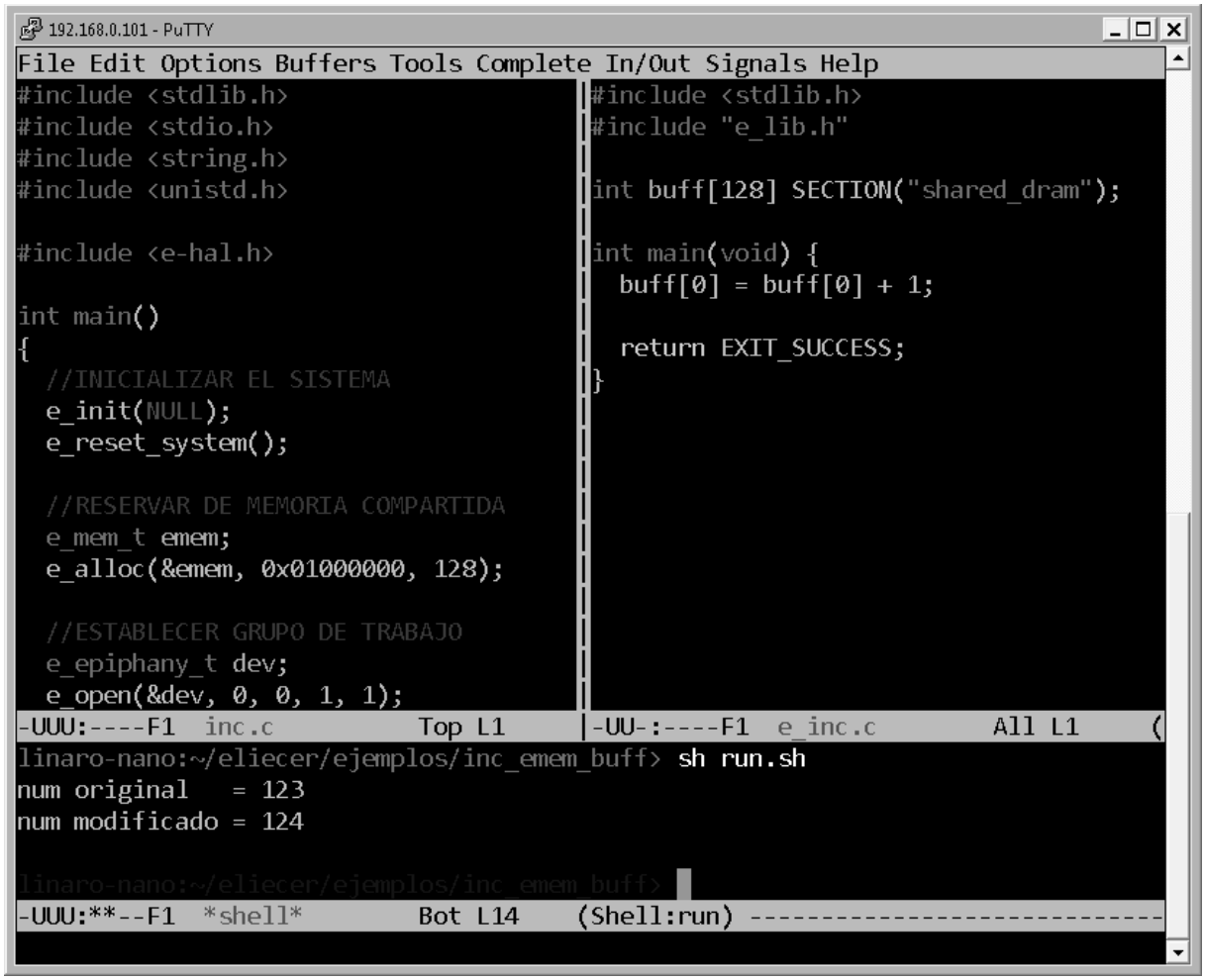


Ilustración 7-17. Utilizando Emacs (modo texto) desde PuTTY

7.5 Software utilizado

El ordenador del laboratorio es un Core 2 Quad a 2.4 GHz con una RAM de 4GB. Utiliza un sistema operativo **Windows 7 de 32 bits**. Para conectar con Parallella-16 se ha utilizado **PuTTY 0.62** y **WinSCP 5.5.5**.

El sistema operativo Linux de **Parallella-16** ha sido **Linaro Nano 3.12**, que venía con el **eSDK 5.13.09**. El servidor VNC utilizado ha sido **vnc4server 4.1.1** y como editor de texto **Emacs 24.3.1**.

Las ocasiones en las que he tenido que estar presente en el laboratorio han sido escasas. He trabajado en remoto desde casa, unas veces desde un PC y otras desde el portátil. La conexión mediante ***Conexión a Escritorio remoto*** me ha permitido trabajar igual en Windows que en Linux. La ilustración 7-5 se puede ver la conexión desde Windows 7, y la ilustración 7-6 desde Ubuntu (Linux). Utilizar en casa Linux en lugar de Windows me ha permitido utilizar ***Remmina***, una aplicación para conexión remota que, además de permitir conectar por escritorio remoto al ordenador del laboratorio, permite conectar a Parallella-16 utilizando el ordenador del laboratorio como túnel SSH, lo cual evita la situación incómoda que se puede ver en la siguiente ilustración.

Ilustración 7-18. Parallella-16 desde Windows 7 con *Conexión a Escritorio remoto*

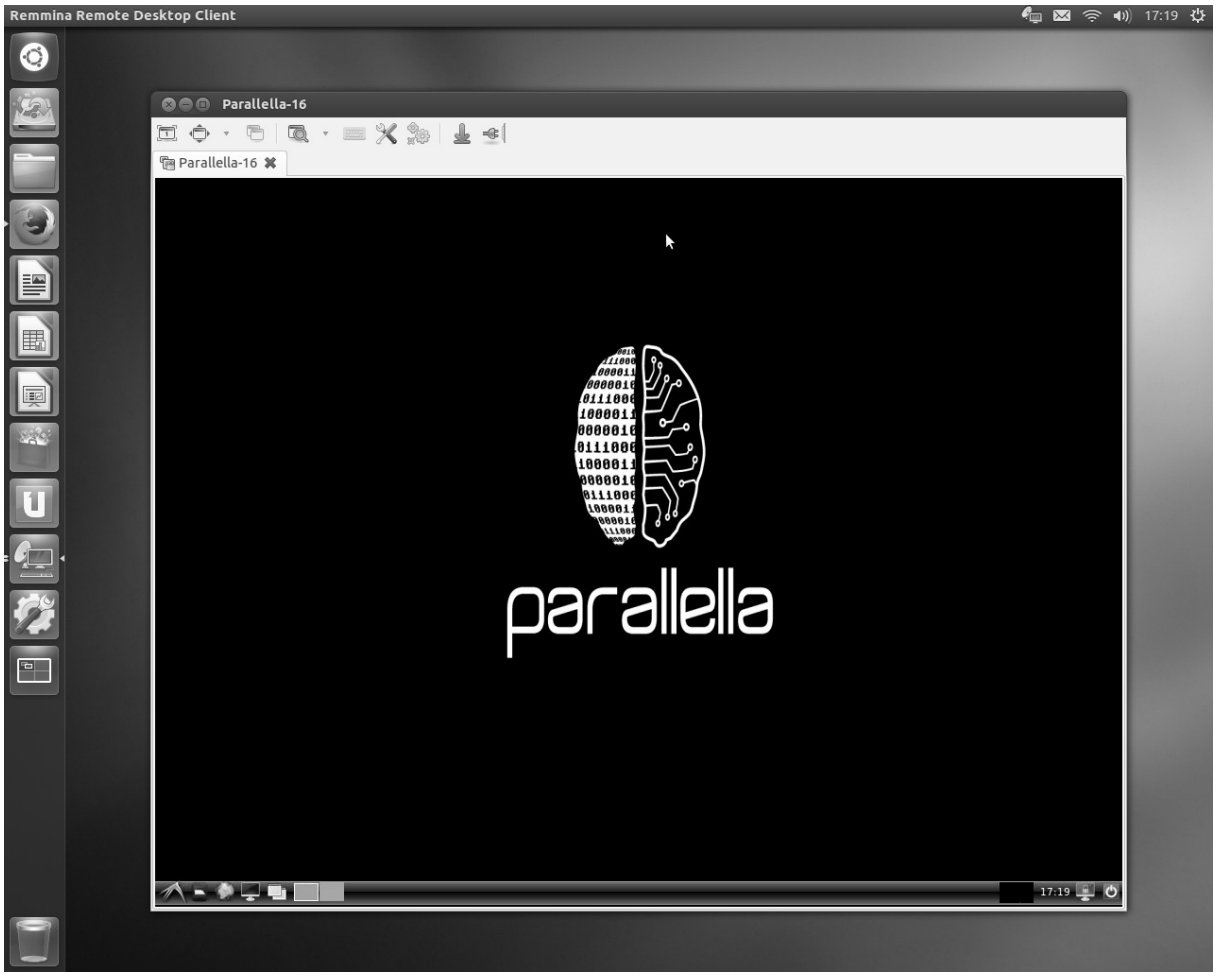


Ilustración 7-19. Parallella-16 desde Ubuntu con *Remmina*

8 EPIPHANY HOST LIBRARY (EHAL)

En este capítulo se presentan las funciones y las estructuras de datos aportadas por la librería *e-hal.h* (Epiphany Hardware Abstraction Layer). Esta librería se utiliza para programar los ejecutables que correrán en el *host*, el procesador ARM. La librería utilizada para programar los ejecutables de los núcleos es *e_lib.h*.

Se trata de una traducción y ampliación del mismo capítulo que podemos encontrar en el manual *Epiphany SDK Reference* que acompaña al eSDK.

8.1 Introducción

Además de utilizar la librería para programar, el compilador y el enlazador deben configurarse con las rutas a los archivos de cabecera y las librerías binarias.

```
$ gcc -I${EPIPHANY_HOME}/tools/host/include \

      -L${EPIPHANY_HOME}/tools/host/lib -le-hal ...
```

Modo básico de operación

El modo estándar de operación es trabajar con grupos de trabajo. Estos grupos son mallas rectangulares de núcleos que se establecen para realizar una tarea. Es posible cargar todos los núcleos del grupo de trabajo con la misma copia del ejecutable (estilo SPMD), o cargar subgrupos, o incluso cargar cada núcleo diferentes con ejecutables diferentes. Es responsabilidad del usuario asegurarse de que las tareas no se asignan a núcleos que ya están siendo utilizados.

Arquitectura de la memoria externa (compartida)

La aplicación principal en el *host* puede comunicarse con los núcleos accediendo a la memoria local del núcleo o mediante un buffer a la memoria externa (compartida) del dispositivo.

En el capítulo dedicado a la arquitectura de la plataforma, se ha explicado cómo la memoria a la que acceden es la misma (la memoria DRAM del host), pero lo hacen utilizando vías distintas (el host con un bus del sistema y los núcleos por eLinks), y por esta razón, aunque tengan cada uno un puntero a la misma zona de memoria, sus direcciones pueden ser diferentes al estar mapeadas.

Las direcciones de base (real y alias) de la memoria externa (compartida) se encuentran en el archivo HDF (*Hardware Description File*). La dirección base alias también se define en el archivo LDF (*Linker Description File*).

Para trabajar con valores lógicos, ya que C no cuenta con uno, se ha definido el siguiente tipo:

```
typedef enum {
     E_FALSE,
     E_TRUE,
} e_bool_t;
```

Y para indicar el resultado de la ejecución de una función, algunas devuelven el siguiente tipo:

```
typedef enum {
     E_OK,
     E_ERR,
     E_WARN,
} e_return_stat_t;
```

Los registros con los que cuenta el sistema son:

```
// General Purpose Registers
// (see Epiphany Architecture Manual for details)
typedef enum
{
     E_REG_R0,       E_REG_R8,       E_REG_R16,      E_REG_R24,
     E_REG_R1,       E_REG_R9,       E_REG_R17,      E_REG_R25,
     E_REG_R2,       E_REG_R10,      E_REG_R18,      E_REG_R26,
     E_REG_R3,       E_REG_R11,      E_REG_R19,      E_REG_R27,
     E_REG_R4,       E_REG_R12,      E_REG_R20,      E_REG_R28,
     E_REG_R5,       E_REG_R13,      E_REG_R21,      E_REG_R29,
     E_REG_R6,       E_REG_R14,      E_REG_R22,      E_REG_R30,
     E_REG_R7,       E_REG_R15,      E_REG_R23,      E_REG_R31,

     E_REG_R32,      E_REG_R40,      E_REG_R48,      E_REG_R56,
     E_REG_R33,      E_REG_R41,      E_REG_R49,      E_REG_R57,
     E_REG_R34,      E_REG_R42,      E_REG_R50,      E_REG_R58,
     E_REG_R35,      E_REG_R43,      E_REG_R51,      E_REG_R59,
     E_REG_R36,      E_REG_R44,      E_REG_R52,      E_REG_R60,
     E_REG_R37,      E_REG_R45,      E_REG_R53,      E_REG_R61,
     E_REG_R38,      E_REG_R46,      E_REG_R54,      E_REG_R62,
     E_REG_R39,      E_REG_R47,      E_REG_R55,      E_REG_R63,
} e_gp_reg_id_t;
```

```
// eCore Special Registers
typedef enum
{
     // DMA registers
     E_REG_DMA0CONFIG,     E_REG_DMA1CONFIG,
     E_REG_DMA0STRIDE,     E_REG_DMA1STRIDE,
     E_REG_DMA0COUNT,      E_REG_DMA1COUNT,
     E_REG_DMA0SRCADDR,    E_REG_DMA1SRCADDR,
     E_REG_DMA0DSTADDR,    E_REG_DMA1DSTADDR,
     E_REG_DMA0AUTODMA0,   E_REG_DMA1AUTODMA0,
     E_REG_DMA0AUTODMA1,   E_REG_DMA1AUTODMA1,
     E_REG_DMA0STATUS,     E_REG_DMA1STATUS,

     // Event Timer Registers
     E_REG_CTIMER0,
     E_REG_CTIMER1,

     // Control Registers
     E_REG_CONFIG,         E_REG_IRET,           E_REG_LC,
     E_REG_STATUS,         E_REG_IMASK,          E_REG_LS,
     E_REG_FSTATUS,        E_REG_ILAT,           E_REG_LE,
     E_REG_PC,             E_REG_ILATST,
     E_REG_DEBUGSTATUS,    E_REG_ILATCL,
     E_REG_DEBUGCMD,       E_REG_IPEND,


     // Processor Control Registers
     E_REG_MEMPROTECT,
     E_REG_MESH_CONFIG,
     E_REG_COREID,
     E_REG_CORE_RESET,
} e_core_reg_id_t;
```

```
// Chip Registers
// (see Epiphany Chip Datasheets for details)
typedef enum
{
      E_REG_IO_LINK_MODE_CFG,
      E_REG_IO_LINK_TX_CFG,
      E_REG_IO_LINK_RX_CFG,
      E_REG_IO_LINK_DEBUG,
      E_REG_IO_GPIO_CFG,
      E_REG_IO_FLAG_CFG,
      E_REG_IO_SYNC_CFG,
      E_REG_IO_HALT_CFG,
      E_REG_IO_RESET,
} e_chip_reg_id_t;

// Epiphany system registers
// (see Board manual for details)
typedef enum
{
      E_SYS_CONFIG,
      E_SYS_RESET,
      E_SYS_VERSION,
      E_SYS_FILTERL,
      E_SYS_FILTERH,
      E_SYS_FILTERC,
} e_sys_reg_id_t
```

8.2 Funciones de Configuración de la Plataforma

Estas funciones son utilizadas para inicializar y preparar el sistema Epiphany para trabajar con él desde el host. También se utilizan para consultar y recuperar información de la plataforma:

- e_init()
- e_get_platform_info()
- e_finalize()

8.2.1 e_init()

Definición

```
int e_init(
      char  *hdf
);
```

Descripción

Esta función inicializa las estructuras de datos de HAL (*Hardware Abstraction Layer*), y establece la conexión con la plataforma Epiphany. Los parámetros de la plataforma se leen desde un archivo HDF (*Hardware Description File*), cuya ruta se da como argumento de la función.

Si se pasa como parámetro `NULL`, la función toma como ruta del archivo HDF la devuelta por la variable de entorno *EPIPHANY_HDF*. Esta variable está normalmente en el archivo de arranque (~/.bashrc), y refleja la estructura de datos de la plataforma Epiphany.

Valor devuelto

Si la inicialización tiene éxito devuelve `E_OK`. En caso contrario `E_ERR`.

Ejemplos

Todos los ejemplos hacen uso de esta función.

Nota: En el momento del lanzamiento, el analizador XML no estaba todavía plenamente integrado en el controlador. En lugar de un archivo de descripción XML, la biblioteca utiliza un simple archivo de texto para listar los componentes de la plataforma. Por favor, use los archivos proporcionados o cree el suyo propio:

EPIPHANY_HDF = "${EPIPHANY_HOME}/bsps/parallella/parallella.hdf"

Podemos consultar el valor de esta variable de entorno escribiendo:

```
Linaro:~> echo $EPIPHANY_HDF
/opt/adapteva/esdk/bsps/current/plataform.hdf
```

8.2.2 e_get_platform_info()

Definición

```
int e_get_platform_info(
     e_platform_t *platform
);
```

Descripción

La información de la plataforma Epiphany está grabada internamente en un objeto de tipo e_platform_t. Contiene los datos sobre los diferentes chips, los segmentos de memoria externos y la geometría del sistema. Algunos de estos datos pueden ser recuperados a través de esta función.

Valor devuelto

Si la inicialización tiene éxito devuelve `E_OK`. En caso contrario `E_ERR`.

Ejemplos

`/info` – En este ejemplo se muestra cómo consultar los datos de la plataforma.

```
VERSION   : PARALLELLA1601

ROW       : 32
COL       : 8

ROWS      : 4
COLS      : 4

NUM_CHIPS : 1

NUM_EMEMS : 1
```

8.2.3 e_finalize()

Definición

```
int e_finalize();
```

Descripción

Esta función finaliza la conexión con el sistema Epiphany. Los recursos que reservados con la función *e_init()* son liberados.

Valor devuelto

Si la inicialización tiene éxito devuelve `E_OK`. En caso contrario `E_ERR`.

Ejemplos

Todos los ejemplos hacen uso de esta función.

8.3 Funciones para Grupos de Trabajo y Memoria Externa

Estas funciones se utilizan para crear grupos de trabajo y asignar memoria externa a los buffers:

- e_open()
- e_close()
- e_alloc()
- e_free()

8.3.1 e_open()

Definición

```
int e_open(
     e_epiphany_t  *dev,
     unsigned     row,
     unsigned     col,
     unsigned     rows,
     unsigned     cols
);
```

Descripción

Esta función define un grupo de trabajo. El grupo de trabajo se define en términos de las coordenadas relativas a la malla 2D que forman los núcleos. Los argumentos `row` y `col` son las coordenadas relativas al primer núcleo, el situado más arriba a la izquierda del grupo de trabajo que se quiere definir.

Los argumentos `rows` y `cols` indican cuantas filas y columnas tiene el grupo de trabajo. Un grupo de trabajo puede contener un solo núcleo, como en el ejemplo *hello-world*.

El grupo de trabajo es almacenado en el objeto `dev` de tipo `e_epiphany_t`. Los accesos posteriores al grupo de trabajo (por ejemplo, para la leer y escribir datos) se realizan utilizando una la variable `dev`, nunca haciendo referencia al núcleo de forma individual.

(32, 8)	(32, 9)	(32, 10)	(32, 11)
(33, 8)	(33, 9)	(33, 10)	(33, 11)
(34, 8)	(34, 9)	(34, 10)	(34, 11)
(35, 8)	(35, 9)	(35, 10)	(35, 11)

Ejemplo

Para definir el grupo de trabajo en gris, no se utilizan las coordenadas reales (33,9), sino las coordenadas relativas (1,1).

Los valores de `rows` y `cols`, que indican cuantas filas y columnas tienen el grupo de trabajo, son 3 y 3.

Valor devuelto

Si la función tiene éxito devuelve `E_OK`. En caso contrario devuelve `E_ERR`.

Ejemplos

Cualquier ejemplo que haga uso de los núcleos utiliza esta función para crear el grupo de trabajo que los contiene.

8.3.2 e_close()

Definición

```
int e_close(
     e_epiphany_t *dev
);
```

Descripción

Esta función cierra, o disuelve, un grupo de trabajo. Los recursos reservados al utilizar la función *e_open()* son liberados. Debe utilizarse esta función antes de reutilizar un núcleo en un nuevo grupo de trabajo, como en el ejemplo *hello-world*.

Valor devuelto

Si la función tiene éxito devuelve `E_OK`. En caso contrario devuelve `E_ERR`.

Ejemplos

Cualquier ejemplo que haga uso de los núcleos utiliza esta función para cerrar, o disolver, el grupo de trabajo.

8.3.3 e_alloc()

Definición

```
int e_alloc(
      e_mem_t    *mbuf,
      off_t      base,
      size_t     size
);
```

Descripción

Esta función reserva espacio en la memoria externa (compartida) para un buffer. El buffer es un objeto del tipo `e_emem_t`. El tamaño del buffer viene determinado por `size`. El desplazamiento (offset) sobre la memoria externa viene determinado por el argumento `base`. El acceso al buffer, para leer o escribir en él, se hará utilizando el objeto `mbuf`.

La dirección de la memoria externa (compartida) se define en la llamada *e_init()*, y el inicio de esta memoria externa corresponde aquí al offset 0.

Valor devuelto

Si la función tiene éxito devuelve `E_OK`. En caso contrario devuelve `E_ERR`.

Ejemplos

En este ejemplo se utiliza esta función para reservar la memoria compartida:

`/inc_emem_buff` – En este ejemplo el host coloca un número en la memoria externa (compartida) y un núcleo accede a esta memoria para incrementarlo. El siguiente ejemplo es muy parecido.

`/inc_emem_func` – Este ejemplo hace exactamente igual que el anterior, pero el núcleo, en lugar de declarar una variable en la memoria compartida, utiliza una función para leer la memoria compartida. Son dos formas de acceder a la memoria compartida.

`/whoiam_emem` – En este ejemplo el host recibe de un núcleo los datos que este puede ver relacionados con el grupo de trabajo al que pertenece utilizando la memoria compartida.

Hay que recordar que, como el sistema reserva 32 MB para código y datos, siendo los primeros 16 MB para código y los siguientes 16 MB para datos, los ejemplos que utilizan un offset (0x1000000) saltar los primeros 16 MB y acceder al principio de la memoria reservada para datos.

8.3.4 e_free()

Definición

```
int e_free(
      e_mem_t   *mbuf
);
```

Descripción

Esta función libera los recursos reservados al utilizar la función *e_alloc()*. Debe utilizar esta función antes de volver a reservar este espacio en la memoria externa para un nuevo buffer.

Valor devuelto

Si la función tiene éxito devuelve `E_OK`. En caso contrario devuelve `E_ERR`.

Ejemplos

Al final de estos ejemplos se utiliza esta función para liberar la memoria reservada previamente con *e_alloc()*.

`/inc_emem_buff` – En este ejemplo el host coloca un número en la memoria externa (compartida) y un núcleo accede a esta memoria para incrementarlo.

`/inc_emem_func` – Este ejemplo hace exactamente igual que el anterior, pero el núcleo, en lugar de declarar una variable en la memoria compartida, utiliza una función para leer la memoria compartida. Son dos formas de acceder a la memoria compartida.

`/whoiam_emem` – En este ejemplo el host recibe de un núcleo los datos que este puede ver relacionados con el grupo de trabajo al que pertenece.

8.4 Funciones de Transferencia de Datos

Estas funciones son utilizadas para leer y escribir con los buffer sobre la memoria externa (compartida) y sobre los núcleos de un grupo de trabajo:

- e_read()
- e_write()

8.4.1 e_read()

Definición

```
ssize_t e_read(
     void       *dev,
     unsigned   row,
     unsigned   col,
     off_t      from_addr,
     void       *buf,
     size_t     size
);
```

Descripción

Esta función lee un dato de tamaño `size` y lo almacena en el buffer local `buf`. El argumento `from_addr` especifica el offset.

Cuando se quiera leer la memoria o los registros de un núcleo, el argumento `dev` será del tipo `e_epiphany_t`, utilizado para definir un grupo de trabajo. El núcleo será aquél con coordenadas (`row`, `col`) dentro del grupo de trabajo.

Cuando se quiera leer de la memoria externa (compartida) el argumento `dev` será del tipo `e_mem_t`. Los argumentos `row` y `col` serán ignorados.

Para acceder a los registros del sistema el argumento `from_addr` deberá ser del tipo `e_gp_reg_id_t`, `e_core_reg_id_t`, `e_chip_reg_id_t`, o `e_sys_reg_id_t`.

Valor devuelto

La función devolverá el número de bytes leídos. En otro caso `E_ERR`.

Ejemplos

`/whoiam_emem` – En este ejemplo utiliza la memoria externa (compartida) para que el núcleo comunique sus datos al host.

`/whoiam_bank` – Es el mismo ejemplo que el anterior, pero se utiliza el banco de memoria del núcleo para comunicar los datos al host.

`/whoiam_reg` – Es el mismo ejemplo que los anteriores, pero utiliza los registros `R32` a `R39` para que el núcleo comunique sus datos al host.

SIEMPRE QUE SE PUEDA HAY QUE EVITAR UTILIZAR LOS REGISTROS PARA COMUNICAR DATOS, YA QUE SON UTILIZADAS INTERNAMENTE POR LA

PLATAFORMA Y SU MANIPULACIÓN INCORRECTA PUEDE HACER QUE SU FUNCIONAMIENTO SEA INCORRECTO.

8.4.2 e_write()

Definición

```
ssize_t e_write(
      void            *dev,
      unsigned        row,
      unsigned        col,
      off_t           to_addr,
      const void      *buf,
      size_t          size
);
```

Descripción

Esta función escribe un dato de tamaño `size` y almacenado en el buffer local `buf`. El argumento `from_addr` especifica el offset.

Cuando se quiera escribir la memoria o los registros de un núcleo, el argumento `dev` será del tipo `e_epiphany_t`, utilizado para definir un grupo de trabajo. El núcleo será aquél cuyas coordenadas dentro del grupo de trabajo sean (`row`, `col`).

Cuando se quiera escribir en la memoria externa el argumento `dev` será del tipo `e_mem_t`. En este caso los argumentos `row` y `col` son ignorados.

Para acceder a los registros del sistema el argumento `from_addr` deberá ser del tipo `e_gp_reg_id_t`, `e_core_reg_id_t`, `e_chip_reg_id_t`, o `e_sys_reg_id_t`.

Valor devuelto

Si la escritura tiene éxito, el número de bytes escritos. En otro caso `E_ERR`.

Ejemplos

`/inc_emem_buff` – En este ejemplo el host coloca un número en la memoria externa (compartida) y un núcleo accede a esta memoria para incrementarlo.

`/inc_emem_func` – Este ejemplo hace exactamente igual que el anterior, pero el núcleo, en lugar de declarar una variable en la memoria compartida, utiliza una función para leer la memoria compartida.

`/inc_bank` – Es el mismo ejemplo que los anteriores, pero utiliza uno de los bancos de memoria para recibir e incrementar el número pasado por el *host*.

SIEMPRE QUE SE PUEDA HAY QUE EVITAR UTILIZAR LOS REGISTROS PARA COMUNICAR DATOS, YA QUE SON UTILIZADAS INTERNAMENTE POR LA PLATAFORMA Y SU MANIPULACIÓN INCORRECTA PUEDE HACER QUE SU FUNCIONAMIENTO SEA INCORRECTO.

8.5 Funciones de Control del System

Estas funciones permiten controlar diferentes aspectos del sistema y la ejecución de un programa:

- e_reset_system()
- e_reset_core()
- e_start()
- e_start_group()
- e_signal()
- e_halt()
- e_resume()

8.5.1 e_reset_system()

Definición

```
int e_reset_system();
```

Descripción

Esta función sirve para poner a punto todo el hardware de la plataforma, incluyendo los chips de la Epifanía y la lógica FPGA.

Se debe tener especial cuidado cuando se utiliza esta función en un entorno de multiprocesamiento para no interrumpir la ejecución de tareas, posiblemente lanzadas por otras aplicaciones.

Valor devuelto

Si la función tiene éxito devuelve `E_OK`. En caso contrario devuelve `E_ERR`.

Ejemplos

Todos los ejemplos hacen uso de esta función.

8.5.2 e_start()

Definición

```
int e_start(
     e_epiphany_t    *dev,
     unsigned        row,
     unsigned        col
);
```

Descripción

Esta función sirve para poner en marcha los núcleos cuando estos no han arrancado o han sido detenidos. Los parámetros `row` y `col` especifican las coordenadas del núcleo relativas al grupo de trabajo `dev`.

Desde un punto de vista más técnico, esta función escribe la señal SYNC en el registro ILAT de un núcleo. Normalmente, esto se utilizará después de cargar un archivo ejecutable en el núcleo. Esto causa que el núcleo salte a la entrada IVT número 0.

Valor devuelto

Si la función tiene éxito devuelve `E_OK`. En caso contrario devuelve `E_ERR`.

Ejemplos

`/inc_emem_buff` – En este ejemplo se reserva memoria compartida para que el host pase un número a un núcleo y que este lo incremente. Se utiliza esta función para iniciar la ejecución del programa en el núcleo una vez que el *host* ha escrito en la memoria compartida. Si lo hiciera antes, la carga del ejecutable borraría la memoria y el dato no llegaría al núcleo.

`/inc_emem_func` – Este ejemplo hace exactamente igual que el anterior, pero el núcleo, en lugar de declarar una variable en la memoria compartida, utiliza una función para leer la memoria compartida.

8.5.3 e_start_group()

Definición

```
int e_start_group(
      e_epiphany_t *dev
);
```

Descripción

Esta función arranca todos los núcleos del grupo de trabajo dev.

Un motivo por el que la ejecución de los programas en los núcleos se retrase (indicando explícitamente que no se ejecuten) puede ser porque se quiere pasar datos por la memoria externa (compartida), ya que esta memoria parece ser borrada al cargar los ejecutables.

Valor devuelto

Si la función tiene éxito devuelve `E_OK`. En caso contrario devuelve `E_ERR`.

Ejemplos

`/mutex` – En este ejemplo varios núcleos intentan incrementar un número almacenado en la memoria externa (compartida). Para que los núcleos puedan leer los datos en la memoria externa debe escribirlos una vez que los núcleos han sido cargados. Esta función se utiliza en este ejemplo para arrancar la ejecución de todos los núcleos del grupo de trabajo.

8.5.4 e_signal()

Definición

```
int e_signal(
     e_epiphany_t   *dev,
     unsigned      row,
     unsigned      col
);
```

Descripción

Esta función escribe la señal USER_INT en el registro ILAT de un núcleo del grupo de trabajo. Esto causa que el núcleo modifique el bit 9 de su IVT (Interrupt Vector Table).

Los parámetros `row` y `col` especifican las coordenadas del núcleo relativas al grupo de trabajo `dev`.

Valor devuelto

Si la función tiene éxito devuelve E_OK. En caso contrario devuelve E_ERR.

Ejemplos

Ningún ejemplo hace uso de esta función.

El registro ILAT anota todas las interrupciones de eventos. Cada bit del registro ILAT (excepto el del bit 9) está ligado a un evento específico del hardware. Se puede acceder a este registro directamente o a través de sus dos alias ILATST y ILATCL.

8.5.5 e_halt()

Definición

```
int e_halt(
     e_epiphany_t   *dev,
     unsigned       row,
     unsigned       col
);
```

Descripción

Esta función detiene la ejecución de un núcleo. Esto puede ser útil a la hora de depurar la aplicación. Para reanudar su ejecución se utiliza *e_resume()*.

Los parámetros `row` y `col` especifican las coordenadas del núcleo relativas al grupo de trabajo `dev`.

Valor devuelto

Si la función tiene éxito devuelve `E_OK`. En caso contrario devuelve `E_ERR`.

Ejemplos

`/halt` – En este ejemplo un núcleo incremente indefinidamente un número almacenado en su banco de memoria. Se utiliza esta función para detener el núcleo antes de leer su valor.

8.5.6 e_resume()

Definición

```
int e_resume(
     e_epiphany_t    *dev,
     unsigned        row,
     unsigned        col
);
```

Descripción

Esta función reanuda la ejecución de un núcleo que ha sido detenido previamente con la función *e_halt()*.

Los parámetros `row` y `col` especifican las coordenadas del núcleo relativas al grupo de trabajo `dev`.

Valor devuelto

Si la función tiene éxito devuelve `E_OK`. En caso contrario devuelve `E_ERR`.

Ejemplos

`/halt` – En este ejemplo un núcleo incremente indefinidamente un número almacenado en su banco de memoria. Se utiliza esta función para reanudar la ejecución del programa en el núcleo, detenido antes de leer el valor del número.

8.6 Funciones de Carga de Programas

Estas funciones sirven para cargar un programa en uno o varios núcleos de un grupo de trabajo. Opcionalmente, los programas cargados pueden iniciarse inmediatamente después de su cargar.

- e_load()
- e_load_group()

8.6.1 e_load()

Definición

```
int e_load(
     char          *executable,
     e_epiphany_t  *dev,
     unsigned      row,
     unsigned      col,
     e_bool_t      start
);
```

Descripción

Esta función carga un programa en un núcleo. La cadena `executable` especifica la ruta del programa. Los parámetros `row` y `col` especifican las coordenadas del núcleo relativas al grupo de trabajo `dev`.

Opcionalmente, un programa cargado se puede iniciar inmediatamente después de la carga, de acuerdo con el parámetro `start`. Cuando el valor de `start` es `E_TRUE`, el programa se pone en marcha después de la carga. Si su valor es `E_FALSE`, no se pone en marcha el programa.

La carga de un programa se debe realizar sólo cuando el núcleo se encuentra parado o en un estado de inactividad. Una manera segura de lograr esto es utilizar *e_reset_system()* o *e_reset_core()* antes de la carga.

Valor devuelto

Si la función tiene éxito devuelve `E_OK`. En caso contrario devuelve `E_ERR`.

Ejemplos

Todos los ejemplos que hacen uso de un núcleo utilizan esta función, o de la función *e_load_group()*, para cargar en ellos los ejecutables.

En la actualidad, eHAL soporta la carga de archivos ejecutables en formato SREC. Se utiliza la herramienta e-objcopy para generar los archivos SREC desde un archivo ejecutable ELF.

8.6.2 e_load_group()

Definición

```
int e_load_group(
     char          *executable,
     e_epiphany_t  *dev,
     unsigned      row,
     unsigned      col,
     unsigned      rows,
     unsigned      cols,
     e_bool_t      start
);
```

Descripción

Esta función carga un programa en los núcleos (no necesariamente todos) de un grupo de trabajo. La cadena `executable` especifica la ruta del programa.

Los núcleos en los que se cargará el programa son un subgrupo dentro del grupo de trabajo. El núcleo superior izquierdo de este subgrupo viene especificado por las coordenadas relativas al grupo de trabajo (`row`, `col`). Los argumentos `rows` y `cols` especifican el número de filas y columnas del subgrupo.

Opcionalmente, un programa cargado se puede iniciar inmediatamente después de la carga, de acuerdo con el parámetro `start`. Cuando el valor de `start` es `E_TRUE`, el programa se pone en marcha después de la carga. Si su valor es `E_FALSE`, no se pone en marcha el programa.

La carga de un programa se debe realizar sólo cuando el núcleo se encuentra parado o en un estado de inactividad. Una manera segura de lograr esto es utilizar *e_reset_system()* o *e_reset_core()* antes de la carga.

Valor devuelto

Si la función tiene éxito devuelve `E_OK`. En caso contrario devuelve `E_ERR`. Algunos errores no fatales devuelven un valor `E_WARN`. El analizador SREC ignora los errores y continúa la carga del programa.

Ejemplos

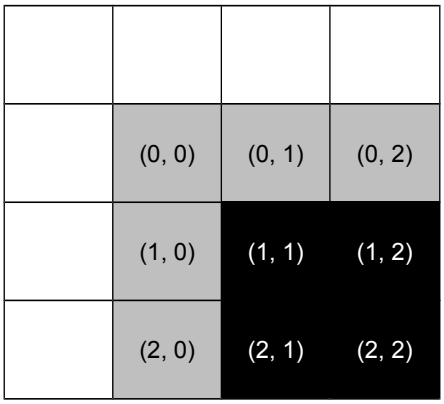
`/mutex` – En este ejemplo varios núcleos intentan incrementar un número almacenado en la memoria externa (compartida). Esta función se utiliza para cargar en todos ellos el mismo ejecutable.

Ejemplo 1

- Núcleo (0, 1)
- Tamaño 1x2

Para cargar un programa en los núcleo en negro del grupo de trabajo en gris habría que utilizar:

- `row : 0`
- `col : 1`
- `rows: 1`
- `cols: 2`



Ejemplo 2

- Núcleo (1, 1)
- Tamaño 2x2

Para cargar un programa en los núcleo en negro del grupo de trabajo en gris habría que utilizar:

- `row : 1`
- `col : 1`
- `rows: 2`
- `cols: 2`

8.7 Funciones de Utilidad

Este es un conjunto de funciones de utilidad, previstas para facilitar algunas tareas de programación en las aplicaciones del host:

- e_get_num_from_coords()
- e_get_coords_from_num()
- e_is_addr_on_chip()
- e_is_addr_on_group()
- e_set_host_verbosity()
- e_set_loader_verbosity()

8.7.1 e_get_num_from_coords()

Definición

```
unsigned e_get_num_from_coords(
      e_epiphany_t  *dev,
      unsigned      row,
      unsigned      col
);
```

Descripción

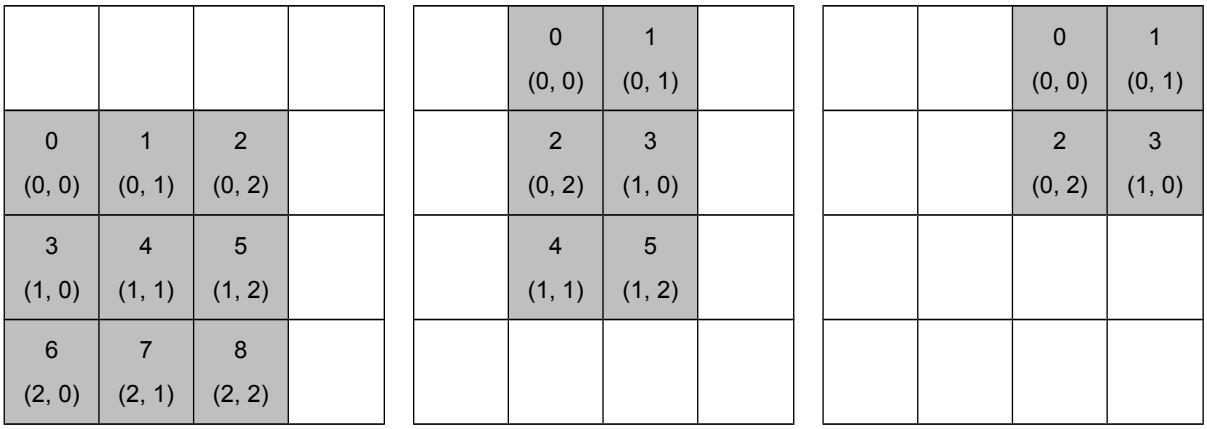
Convierte las coordenadas relativas de un núcleo en un grupo de trabajo a un número. El grupo de trabajo está definido por el argumento `dev`.

Valor devuelto

Esta función devuelve el número asignado al núcleo dentro del grupo de trabajo.

Ejemplos

`/core_number` – En este ejemplo muestra el chip, un grupo de trabajo y el número que cada núcleo tiene asignado dentro de este grupo. Esta función se utiliza para obtener el número asignado a cada núcleo.



8.7.2 e_get_coords_from_num()

Definición

```
void e_get_coords_from_num(
e_epiphany_t    *dev,
unsigned        corenum,
unsigned        *row,
unsigned        *col
);
```

Descripción

Calcula las coordenadas relativas `row` y `col` del núcleo que en el grupo de trabajo `dev` tiene el asignado el número `corenum`.

Valor devuelto

Ninguno.

Ejemplos

`/core_coords` – En este ejemplo se listan en orden, desde 0, todos los núcleos del grupo de trabajo con sus coordenadas. Esta función se utiliza para obtener las coordenadas que ocupa cada uno de estos núcleos.

0 (0, 0)	1 (0, 1)	2 (0, 2)	
3 (1, 0)	4 (1, 1)	5 (1, 2)	
6 (2, 0)	7 (2, 1)	8 (2, 2)	

	0 (0, 0)	1 (0, 1)	
	2 (0, 2)	3 (1, 0)	
	4 (1, 1)	5 (1, 2)	

		0 (0, 0)	1 (0, 1)
		2 (0, 2)	3 (1, 0)

8.7.3 e_is_addr_on_chip()

Definición

```
e_bool_t e_is_addr_on_chip(
     void  *addr
);
```

Descripción

Esta función comprueba si una dirección global de 32 bits, dada como argumento `addr`, está dentro del espacio de memoria del chip Epifanía.

Valor devuelto

Esta función devuelve `E_TRUE` si la dirección está dentro del chip. En caso contrario devuelve `E_FALSE`.

Ejemplos

`/address` – En este ejemplo se crea un puntero a una zona de memoria y un grupo de trabajo y se indica si el puntero apunta al espacio de memoria del chip y al del grupo de trabajo. Esta función se utiliza para comprobar si apunta al espacio de memoria del chip.

En el ejemplo la dirección 0x80800000 apunta al inicio del núcleo 0x808 en hexadecimal, o 100000001000 en binario. Como los 6 dígitos más altos (100000) corresponden a la fila y los 6 más bajos (001000) a la columna, podemos decir que esta dirección apunta es al inicio del núcleo (32,8). En coordenadas relativas al chip, este núcleo corresponde al (0,0).

Si al crear el grupo de trabajo cambiamos este núcleo y ponemos otro, veremos como el resultado cambia.

En el manual *Epiphany Architecture Reference* se encontrar más información acerca de la distribución de la memoria.

8.7.4 e_is_addr_on_group()

Definición

```
e_bool_t e_is_addr_on_group(
     e_epiphany_t  *dev,
     void          *addr
);
```

Descripción

Esta función comprueba si una dirección global de 32 bits, dada como argumento `addr`, está dentro del espacio de memoria de un grupo de trabajo. El grupo de trabajo viene especificado por `dev`.

Valor devuelto

Esta función devuelve `E_TRUE` si la dirección está dentro del grupo de trabajo. En caso contrario devuelve `E_FALSE`.

Ejemplos

`/address` – En este ejemplo se crea un puntero a una zona de memoria y un grupo de trabajo y se indica si el puntero apunta al espacio de memoria del chip y al del grupo de trabajo. Esta función se utiliza para comprobar si apunta al espacio de memoria del grupo de trabajo.

En el ejemplo la dirección 0x80800000 apunta al inicio del núcleo 0x808 en hexadecimal, o 100000001000 en binario. Como los 6 dígitos más altos (100000) corresponden a la fila y los 6 más bajos (001000) a la columna, podemos decir que esta dirección apunta es al inicio del núcleo (32,8). En coordenadas relativas al chip, este núcleo corresponde al (0,0).

En el manual *Epiphany Architecture Reference* se encontrar más información acerca de la distribución de la memoria.

8.7.5 e_set_host_verbosity()

Definición

```
e_hal_diag_t e_set_host_verbosity(
     e_hal_diag_t   verbose
);
```

Descripción

Esta función ajusta el nivel de detalle de funciones en el ejecutable del host. Los niveles definidos van desde `H_D0` a `H_D4`. El nivel `H_D0` no conlleva ningún tipo de diagnóstico. A partir del nivel `H_D1` los diagnósticos son más detallados. Esta función está destinada al diagnóstico y a fines de depuración.

Valor devuelto

Esta función devuelve el nivel de diagnóstico antiguo.

Ejemplos

Ningún ejemplo hace uso de esta función.

8.7.6 e_set_loader_verbosity()

Definición

```
e_loader_diag_t e_set_loader_verbosity(
     e_loader_diag_t   verbose
);
```

Descripción

Esta función ajusta el nivel de detalle de las funciones del ejecutable cargado en los núcleos, por encima de las funciones del ejecutable del *host*. Los niveles definidos van desde `H_D0` a `H_D4`. El nivel `H_D0` no conlleva ningún tipo de diagnóstico. A partir del nivel `H_D1` los diagnósticos son más detallados. Esta función está destinada al diagnóstico y a fines de depuración.

Valor devuelto

Esta función devuelve el nivel de diagnóstico antiguo.

Ejemplos

Ningún ejemplo hace uso de esta función.

9 EPIPHANY HARDWARE UTILITY LIBRARY (ELIB)

En este capítulo se presentan las funciones y las estructuras de datos aportadas por la librería *e_lib.h*. Esta librería se utiliza para programar los ejecutables que correrán en los núcleos. La librería utilizada para programar los ejecutables del *host* (el procesador ARM) es *e-hal.h*.

Se trata de una traducción ampliada del mismo capítulo que podemos encontrar en el manual *Epiphany SDK Reference* que acompaña al eSDK.

9.1 Introducción

Esta librería está compuesta de funciones que permiten configurar y consultar el hadrware de la arquitectura Epiphany. Estas funciones permiten automatizar muchas de las tareas de programación comunes que no son proporcionadas por los lenguajes C/C++ al tratarse de tareas específicas de la arquitectura Epiphany.

En las siguientes secciones se describen las funciones de esta librería, denominada eLib en la documentación oficial. Estas funciones están divididas por familias, según su cometido. El archivo de cabecera de esta librería es "*e_lib.h*", y contiene otros archivos de cabecera correspondientes a cada una de las familias en las que está dividida la API.

El código de las aplicaciones diseñadas para ejecutarse en los núcleos deberán añadir la siguiente línea al inicio del código.

```
#include "e_lib.h"
```

También será necesario utilizar con el compilador *e-gcc* la opción *-le-lib*, con el fin de que este utilice la librería al construir el ejecutable.

Como ya se ha dicho en el capítulo relacionado con la arquitectura, para utilizar los núcleos hay que crear un grupo de trabajo, un subconjunto de la malla 2D. Para que el núcleo conozca ciertos datos relacionados con la memoria y qué posición ocupa dentro del grupo de trabajo al que pertenece, cada núcleo tiene acceso a dos objetos globales.

Uno llamado `e_emem_config` que contiene información sobre la dirección base de la memoria externa.

```
e_emem_config.base          Dirección absoluta de la memoria.
```

El otro llamado `e_group_config` que contiene la información sobre el tipo de chip, la posición y el tamaño de su grupo de trabajo y la posición que ocupa dentro de él.

```
e_group_config.chiptype    Tipo de chip
e_group_config.group_id    ID del primer núcleo
e_group_config.group_row   Fila del primer núcleo
e_group_config.group_col   Columna del primer núcleo
e_group_config.group_rows  Número de filas del grupo
e_group_config.group_cols  Número de columnas del grupo
e_group_config.core_row    Fila que ocupa el núcleo
e_group_config.core_col    Columna que ocupa el núcleo
```

Suponiendo que definimos un grupo de trabajo como el marcado en gris, y que consultamos en el núcleo en negro estos datos, esta sería la información obtenida.

(32, 8)	(32, 9)	(32, 10)	(32, 11)
(33, 8)	(33, 9)	(33, 10)	(33, 11)
(34, 8)	(34, 9)	(34, 10)	(34, 11)
(35, 8)	(35, 9)	(35, 10)	(35, 11)

```
e_group_config.chiptype    0
e_group_config.group_id    2121
e_group_config.group_row   33
e_group_config.group_col   11
e_group_config.group_rows  3
e_group_config.group_cols  3
e_group_config.core_row    0
e_group_config.core_col    2
```

Dos de estos valores necesitan una explicación:

- El chiptype puede tener dos valores, 0 cuando tiene 16 núcleos (E16G301) y 1 cuando tiene 64 (E64G401).
- El ID de los núcleos es un número binario de 12 dígitos. Los 6 bits más bajos corresponden a la fila y los otros 6 a la columna. El entero 2121 es el binario 100001001001, donde 001001 (en decimal 9) representan la columna y 100001 (en decimal 33) representan la fila.

Para simplificar la definición de atributos variables en C se han definido:

```
#define ALIGN(x)    __attribute__ ((aligned (x)))

#define PACKED      __attribute__ ((packed))

#define SECTION(x)  __attribute__ ((section (x)))
```

Se puede encontrar documentación relacionada con la creación de atributos variables en esta página: https://gcc.gnu.org/onlinedocs/gcc/Variable-Attributes.html

Para trabajar con valores lógicos, ya que C no cuenta con uno, se ha definido el siguiente tipo:

```
typedef enum {
     E_FALSE,
     E_TRUE,
} e_bool_t;
```

Y para indicar el resultado de la ejecución de una función, algunas devuelven el siguiente tipo:

```
typedef enum {
     E_OK,
     E_ERR,
     E_WARN,
} e_return_stat_t;
```

9.2 Funciones de acceso al sistema de registros

Las funciones de acceso al sistema de registros y que permiten leer y escribir los registros de forma sencilla son:

- e_reg_read()
- e_reg_write()

Los registros con los que cuenta el sistema son:

```
// General Purpose Registers
typedef enum
{
     E_REG_R0,       E_REG_R8,       E_REG_R16,      E_REG_R24,
     E_REG_R1,       E_REG_R9,       E_REG_R17,      E_REG_R25,
     E_REG_R2,       E_REG_R10,      E_REG_R18,      E_REG_R26,
     E_REG_R3,       E_REG_R11,      E_REG_R19,      E_REG_R27,
     E_REG_R4,       E_REG_R12,      E_REG_R20,      E_REG_R28,
     E_REG_R5,       E_REG_R13,      E_REG_R21,      E_REG_R29,
     E_REG_R6,       E_REG_R14,      E_REG_R22,      E_REG_R30,
     E_REG_R7,       E_REG_R15,      E_REG_R23,      E_REG_R31,

     E_REG_R32,      E_REG_R40,      E_REG_R48,      E_REG_R56,
     E_REG_R33,      E_REG_R41,      E_REG_R49,      E_REG_R57,
     E_REG_R34,      E_REG_R42,      E_REG_R50,      E_REG_R58,
     E_REG_R35,      E_REG_R43,      E_REG_R51,      E_REG_R59,
     E_REG_R36,      E_REG_R44,      E_REG_R52,      E_REG_R60,
     E_REG_R37,      E_REG_R45,      E_REG_R53,      E_REG_R61,
     E_REG_R38,      E_REG_R46,      E_REG_R54,      E_REG_R62,
     E_REG_R39,      E_REG_R47,      E_REG_R55,      E_REG_R63,
} e_gp_reg_id_t;
```

```
// eCore Special Registers
typedef enum
{
      // DMA registers
      E_REG_DMA0CONFIG,     E_REG_DMA1CONFIG,
      E_REG_DMA0STRIDE,     E_REG_DMA1STRIDE,
      E_REG_DMA0COUNT,           E_REG_DMA1COUNT,
      E_REG_DMA0SRCADDR,    E_REG_DMA1SRCADDR,
      E_REG_DMA0DSTADDR,    E_REG_DMA1DSTADDR,
      E_REG_DMA0AUTODMA0,   E_REG_DMA1AUTODMA0,
      E_REG_DMA0AUTODMA1,   E_REG_DMA1AUTODMA1,
      E_REG_DMA0STATUS,     E_REG_DMA1STATUS,

      // Event Timer Registers
      E_REG_CTIMER0,
      E_REG_CTIMER1,


      // Control Registers
      E_REG_CONFIG,         E_REG_IRET,           E_REG_LC,
      E_REG_STATUS,         E_REG_IMASK,          E_REG_LS,
      E_REG_FSTATUS,        E_REG_ILAT,           E_REG_LE,
      E_REG_PC,             E_REG_ILATST,
      E_REG_DEBUGSTATUS,    E_REG_ILATCL,
      E_REG_DEBUGCMD,       E_REG_IPEND,

      // Processor Control Registers
      E_REG_MEMPROTECT,
      E_REG_MESH_CONFIG,
      E_REG_COREID,
      E_REG_CORE_RESET,
} e_core_reg_id_t;
```

```
// Chip Registers
typedef enum
{
      E_REG_IO_LINK_MODE_CFG,
      E_REG_IO_LINK_TX_CFG,
      E_REG_IO_LINK_RX_CFG,
      E_REG_IO_LINK_DEBUG,
      E_REG_IO_GPIO_CFG,
      E_REG_IO_FLAG_CFG,
      E_REG_IO_SYNC_CFG,
      E_REG_IO_HALT_CFG,
      E_REG_IO_RESET,
} e_chip_reg_id_t;
```

9.2.1 e_reg_read()

Definición

```
unsigned e_reg_read(
     e_core_reg_id_t reg_id
);
```

Descripción

Esta función lee el valor de uno de los registros del sistema, aunque las pruebas apuntas más a que solo se pueden leer los registros internos del propio núcleo, como el registro que contiene su id que es `E_REG_COREID`.

Valor devuelto

Devuelve el valor actual del registro del sistema identificado por `reg_id`.

Ejemplos

Ningún ejemplo hace uso de esta función.

SIEMPRE QUE SE PUEDA HAY QUE EVITAR UTILIZAR LOS REGISTROS PARA COMUNICAR DATOS, YA QUE SON UTILIZADAS INTERNAMENTE POR LA PLATAFORMA Y SU MANIPULACIÓN INCORRECTA PUEDE HACER QUE SU FUNCIONAMIENTO SEA INCORRECTO.

9.2.2 e_reg_write()

Definición

```
void e_reg_write(
     e_core_reg_id_t  reg_id,
     unsigned         val
);
```

Descripción

Esta función establece el valor del registros del sistema identificado por `reg_id` con el valor `val`.

Valor devuelto

Ninguno.

Ejemplos

`/whoiam_reg` – En este ejemplo un núcleo pasa al host los datos a los que tiene acceso de su grupo de trabajo. En este ejemplo se escriben los datos en los registros `E_REG_R32` a `E_REG_R39` con esta función.

SIEMPRE QUE SE PUEDA HAY QUE EVITAR UTILIZAR LOS REGISTROS PARA COMUNICAR DATOS, YA QUE SON UTILIZADAS INTERNAMENTE POR LA PLATAFORMA Y SU MANIPULACIÓN INCORRECTA PUEDE HACER QUE SU FUNCIONAMIENTO SEA INCORRECTO.

9.3 Funciones del Servicio de Interrupciones

Para este apartado no se han generado ejemplos. La complejidad del tema y la falta de ejemplos ha hecho que finalmente haya invertido más tiempo y esfuerzo en dejar claros temas menos complejos. Básicamente, este apartado es una traducción del que se puede consultar en el manual oficial.

Las funciones del Servicio de Interrupciones permiten generar y controlar interrupciones del sistema. Estas funciones permiten generar interrupciones tanto en un núcleo local como en un núcleo remoto:

- e_irq_attach()
- e_irq_global_mask()
- e_irq_mask()
- e_irq_set()
- e_irq_clear()

Tipos definidos relacionados con estas funciones:

```
typedef void (*sighandler_t)(int);

typedef enum
{
      E_SYNC,
      E_SW_EXCEPTION,
      E_MEM_FAULT,
      E_TIMER0_INT,
      E_TIMER1_INT,
      E_MESSAGE_INT,
      E_DMA0_INT,
      E_DMA1_INT,
      E_USER_INT,
} e_irq_type_t
```

9.3.1 e_irq_attach()

Definición

```
void e_irq_attach(
     e_irq_type_t   irq,
     sighandler_t   handler
);
```

Descripción

Esta función registra `handler` como una función manejadora de interrupciones (ISR, Interrupt Handler Function), para una entrada de la tabla de vectores de interrupción (IVT, Interrupt Vector Table) especificada por el parámetro `irq`.

Usando esta función, se puede asignar y reemplazar una función manejadora de interrupciones (ISR) a un tipo de evento específico en tiempo de ejecución. Para ello se usa una asignación indirecta, la cual puede suponer un retardo en la ejecución de la función si se produce el evento que maneja.

The ISR should be compiled using the interrupt function attribute in order to apply proper entry and exit sequences, guaranteeing safe context switching.

Valor devuelto

Ninguno.

9.3.2 e_irq_global_mask()

Definición

```
void e_irq_global_mask(
     e_bool_t    state
);
```

Descripción

Esta función permite activar y desactivar globalmente las llamadas de interrupción en un núcleo.

Si `state` es `E_TRUE`, se activa el bit GID del registro de estado del núcleo y en consecuencia los eventos de interrupción son enmascarados.

Si `state` es `E_FALSE`, se desactiva el bit GID del registro de estado del núcleo y en consecuencia los eventos de interrupción son comprobados antes que otros mecanismos de enmascaramiento e interrupciones pendientes.

Valor devuelto

Ninguno.

9.3.3 e_irq_mask()

Definición

```
void e_irq_mask(
     e_irq_type_t   irq,
     e_bool_t       state
);
```

Descripción

Esta función permite activar o desactivar un tipo de evento de interrupción, especificado por `irq`, modificando su respectivo bit en el registro IMASK del núcleo.

Si el `state` es `E_TRUE`, los eventos de interrupción de este tipo son enmascarados.

Si el `state` es `E_FALSE`, los eventos de interrupción de este tipo no son enmascarados.

Valor devuelto

Ninguno.

9.3.4 e_irq_set()

Definición

```
void e_irq_set(
     unsigned      row,
     unsigned      col,
     e_irq_type_t  irq
);
```

Descripción

Esta función genera un evento de interrupción activando el bit especificado por `irq` de su registro ILAT. El evento es generado en el núcleo cuyas coordenadas (`row`, `col`) son relativas a su grupo de trabajo.

Valor devuelto

Ninguno.

9.3.5 e_irq_clear()

Definición

```
void e_irq_clear(
     unsigned        row,
     unsigned        col,
     e_irq_type_t    irq
);
```

Descripción

Esta función limpia (elimina) las peticiones de interrupción pendiente, de tipo `irq`, limpiando su bit del registro ILAT. Esta función actuará en el núcleo cuyas coordenadas (`row`, `col`) son relativas a su grupo de trabajo.

Valor devuelto

Ninguno.

9.4 Funciones Timer

Cada núcleo tiene dos contadores Timer. Estas funciones de la interfaz Timer permiten leer, escribir y manipular estos contadores:

- e_ctimer_get()
- e_ctimer_set()
- e_ctimer_start()
- e_ctimer_stop()
- e_wait()

Tipos y constantes definidas relacionados con estas funciones.

```
typedef enum
{
      E_CTIMER_0,
      E_CTIMER_1,
} e_ctimer_id_t;

//Para ver el significado de estos valores
//consultar el manual "Epiphany Architecture Reference"
typedef enum
{
      E_CTIMER_OFF,
      E_CTIMER_CLK,
      E_CTIMER_IDLE,
      E_CTIMER_IALU_INST,
      E_CTIMER_FPU_INST,
      E_CTIMER_DUAL_INST,
      E_CTIMER_E1_STALLS,
      E_CTIMER_RA_STALLS,
      E_CTIMER_EXT_FETCH_STALLS,
      E_CTIMER_EXT_LOAD_STALLS,
} e_ctimer_config_t;
```

```
#define E_CTIMER_MAX
```

9.4.1 e_ctimer_get()

Definición

```
unsigned e_ctimer_get(
      e_ctimer_id_t   timerid
);
```

Descripción

Lee el valor del contador `timerid` del núcleo que hace la llamada. Tenga en cuenta que los contadores decrecen hasta llegar a 0.

Valor devuelto

Devuelve el valor del contador `timerid`.

Ejemplos

`/timers` – En este ejemplo el núcleo utiliza un contador para calcular el tiempo transcurrido en una operación. En este ejemplo se utiliza esta función para obtener el valor del contador utilizado.

9.4.2 e_ctimer_set()

Definición

```
unsigned e_ctimer_set(
      e_ctimer_id_t   timerid,
      unsigned        val
);
```

Descripción

Establece el valor del contador `timerid` a val. Tenga en cuenta que los contadores decrecen hasta llegar a 0. Use la constante `E_CTIMER_MAX` para establecer el máximo valor permitido.

Valor devuelto

Devuelve el nuevo valor del contador `timerid`.

Ejemplos

`/timers` – En este ejemplo el núcleo utiliza un contador para calcular el tiempo transcurrido en una operación. En este ejemplo se utiliza esta función para establecer el valor del contador utilizado.

9.4.3 e_ctimer_start()

Definición

```
unsigned e_ctimer_start(
      e_ctimer_id_t       timerid,
      e_ctimer_config_t   config
);
```

Descripción

Hace que el contador `timerid` empiece a contar, hacia atrás, los eventos especificados por la variable `config`.

La función establece en el registro de configuración `CTIMERxCFG` (la x puede ser 0 o 1, dependiendo del contador) el valor de `config`. Para más detalles consultar el manual *Epiphany Architecture Reference*.

Valor devuelto

Devuelve el valor actual del contador `timerid`.

Ejemplos

`/timers` – En este ejemplo el núcleo utiliza un contador para calcular el tiempo transcurrido en una operación. En este ejemplo se utiliza esta función para iniciar el contador utilizado.

9.4.4 e_ctimer_stop()

Definición

```
unsigned e_ctimer_stop(
     unsigned   timerid
);
```

Descripción

Hace que el contador timerid se detenga.

La función establece en el registro de configuración `CTIMERxCFG` (la x puede ser 0 o 1, dependiendo del contador) al valor `E_CTIMER_OFF`. Para más detalles consultar el manual *Epiphany Architecture Reference*.

Valor devuelto

Devuelve el valor actual del contador.

Ejemplos

`/timers` – En este ejemplo el núcleo utiliza un contador para calcular el tiempo transcurrido en una operación. En este ejemplo se utiliza esta función para detener el contador utilizado.

9.4.5 e_wait()

Definición

```
void e_wait(
     e_ctimer_id_t   timerid,
     unsigned        clicks
);
```

Descripción

Esta función detiene el programa el número de ciclos de reloj especificadas por `clicks`. Para ello, utiliza el contador `timerid`.

Como consecuencia, se anulará cualquier conteo que se esté realizando sobre el contador timerid. Asegúrese de guardar el valor de este contador antes de llamar *e_wait()* si lo necesita posteriormente.

Tenga en cuenta que el tiempo real (el de un reloj de pared) depende de la velocidad de reloj del chip Epifanía.

Valor devuelto

Ninguno.

Ejemplos

El ejemplo que hace uso de esta función es complejo y está relacionado con las funciones DMA. No es recomendable utilizarlo para ver cómo funciona esta función.

Para ver su funcionalidad, podría utilizarse en algunos ejemplos dentro de los bucles de espera, de esa forma, en lugar de consumir ciclos de reloj mientras se cumple una condición, hace una pausa antes de comprobar si debe seguir a la espera.

9.5 Funciones de Movimiento de Datos y DMA

Las funciones DMA controlan dos canales DMA incluidos en cada núcleo. Se utilizan para consultar el estado, la configuración y la copia de memoria usando el motor DMA.

- e_read()
- e_write()
- e_dma_copy()
- e_dma_start()
- e_dma_busy()
- e_dma_wait()
- e_dma_set_desc()

Tipos definidos relacionadas con estas funciones:

```
typedef enum
{
      E_DMA_0,
      E_DMA_1
} e_dma_id_t;

typedef struct
{
      unsigned config;
      unsigned inner_stride;
      unsigned count;
      unsigned outer_stride;
      void *src_addr;
      void *dst_addr;
} e_dma_desc_t;
```

```
typedef enum
{
      E_DMA_ENABLE,
      E_DMA_MASTER,
      E_DMA_CHAIN,
      E_DMA_STARTUP,
      E_DMA_IRQEN,
      E_DMA_BYTE,
      E_DMA_HWORD,
      E_DMA_WORD,
      E_DMA_DWORD,
      E_DMA_MSGMODE,
      E_DMA_SHIFT_SRC_IN,
      E_DMA_SHIFT_DST_IN,
      E_DMA_SHIFT_SRC_OUT,
      E_DMA_SHIFT_DST_OUT,
} e_dma_config_t;
```

El ***acceso directo a memoria*** (**DMA**, del inglés ***Direct Memory Access***) permite acceder a la memoria del sistema sin hacer uso de la CPU, sino del motor DMA. Una transferencia DMA consiste en copiar un bloque de memoria de un dispositivo a otro. Un ejemplo típico es mover un bloque de memoria desde una memoria externa a una interna más rápida. Tal operación no ocupa al procesador y, por ende, éste puede efectuar otras tareas. Las transferencias DMA son esenciales para aumentar el rendimiento de aplicaciones que requieran muchos recursos.

9.5.1 e_read()

Definición

```
void *e_read(
     void         *remote,
     void         *dst,
     unsigned     row,
     unsigned     col,
     const void   *src,
     size_t       bytes
);
```

Descripción

Esta función lee un dato de tamaño bytes y lo almacena en el buffer local `dst`. Esta lectura la puede hacer sobre la memoria externa (compartida) o sobre un núcleo perteneciente a un grupo de trabajo.

Cuando se quiera leer de un núcleo el argumento `remote` será del tipo `e_group_config`. El núcleo a leer será aquél cuyas coordenadas dentro del grupo de trabajo sean (`row`, `col`). Si la dirección `src` es una dirección global, entonces se utiliza sin modificar.

Cuando se quiera leer de la memoria externa el argumento `remote` será del tipo `e_emem_config`. En este caso los argumentos `row` y `col` son ignorados. La dirección `src` es una dirección relativa a la memoria externa.

Valor devuelto

Ninguno.

Ejemplos

`/inc_emem_func` – En este ejemplo se reserva memoria externa para que el host pase un número a un núcleo y que este lo incremente. Se utiliza esta función para leer el dato en la memoria externa.

La aplicación principal utiliza la función *e_alloc()* para crear el buffer `emem`. El sistema reserva 32 MB para código y datos, siendo los primeros 16 MB para código y los siguientes 16 MB para datos. El valor del offset (0x1000000) fuerza a que esta variable esté localizada al principio de la memoria reservada para datos. El núcleo debe utilizar el mismo offset para acceder a los datos.

9.5.2 e_write()

Definición

```
void *e_write(
     void         *remote,
     const void   *src,
     unsigned     row,
     unsigned     col,
     void         *dst,
     size_t       bytes
);
```

Descripción

Esta función escribe un dato de tamaño `bytes` almacenado en el buffer local `src`. Esta escritura la puede hacer sobre la memoria externa o sobre un núcleo perteneciente a un grupo de trabajo.

Cuando se quiera escribir en un núcleo el argumento `remote` será del tipo `e_group_config`. El núcleo será aquél cuyas coordenadas dentro del grupo de trabajo sean (`row`, `col`). Si la dirección `dst` es una dirección global, entonces se utiliza sin modificar.

Cuando se quiera escribir en la memoria externa el argumento `remote` será del tipo `e_emem_config`. En este caso los argumentos `row` y `col` son ignorados. La dirección `dst` es una dirección relativa a la memoria externa.

Valor devuelto

Ninguno.

Ejemplos

`/inc_emem_func` – En este ejemplo se reserva memoria externa (compartida) para que el host pase un número a un núcleo y que este lo incremente. Se utiliza esta función para escribir el dato en la memoria externa.

La aplicación principal utiliza la función *e_alloc()* para crear el buffer `emem`. El sistema reserva 32 MB para código y datos, siendo los primeros 16 MB para código y los siguientes 16 MB para datos. El valor del offset (0x1000000) fuerza a que esta variable esté localizada al principio de la memoria reservada para datos. El núcleo debe utilizar el mismo offset para acceder a los datos.

9.5.3 e_dma_copy()

Definición

```
int e_dma_copy(
     void       *dst,
     void       *src,
     size_t     bytes
);
```

Descripción

Esta función utiliza el canal DMA1 para copiar bytes `bytes` desde el origen `src` al destino `dst`. Si el canal DMA está ocupado se espera hasta que se concluye la transferencia anterior. Después de iniciar el proceso de transferencia DMA el proceso (el programa) espera hasta que termina la transferencia (bloqueo DMA).

Esta función es una alternativa más rápida que la función estándar *memcpy()*. Sin embargo, la utilización de la DMA, tiene algunas limitaciones que la función estándar no impone, como las restricciones en las direcciones de origen y de destino.

Por favor consulte el manual *Epiphany Architecture Reference* para más detalles.

Valor devuelto

Si ha tenido éxito devuelve 0.

Ejemplos

`/dma_copy` – En este ejemplo, el host almacena en la memoria externa (compartida) diez números enteros. Un núcleo copia estos números a su memoria local (más rápida) estos números, los suma y deja el resultado en la memoria externa (compartida). En este ejemplo se utiliza esta función para copiar los números de la memoria externa a la memoria local del núcleo.

9.5.4 e_dma_start()

Definición

```
int e_dma_start(
     e_dma_desc_t    *descriptor,
     e_dma_id_t      chan
);
```

Descripción

Inicia una transacción DMA utilizando el canal `chan` según los datos del parámetro `descriptor`.

Valor devuelto

Si ha tenido éxito devuelve 0.

Ejemplos

Ambos ejemplos hacen lo mismo, pasar una serie de números desde la memoria compartida a un array local, sumarlos y devolver por la memoria compartida el resultado.

`/dma_start_master` – Este ejemplo utiliza un solo núcleo. Este núcleo declara un descriptor en modo MASTER para pasar esos números de la memoria compartida a un array local utilizando esta función, los suma y devuelve el resultado por la memoria compartida.

`/dma_start_slave` – Este ejemplo utiliza 2 núcleos, ambos con ejecutables distintos. Uno de ellos, el transmisor, lee la memoria compartida y escribe cada número en un registro especial, uno a uno, como si sobrescribiera este registro continuamente. El otro núcleo, que ha creado un descriptor en modo SLAVE (en realidad, lo que hace es no declararlo como MASTER) y sin indicar un origen de datos, al ejecutar esta función, a través del registro especial antes comentado, va leyendo cada uno de esos datos. Cuando finaliza esta función, tiene un array con todos los números que el otro núcleo le ha pasado.

9.5.5 e_dma_busy()

Definición

```
int e_dma_busy(
      e_dma_id_t   chan
);
```

Descripción

Permite consultar el estado del canal DMA `chan`.

Valor devuelto

Devuelve 0 si el canal DMA `chan` no está en uso, de lo contrario devolverá estado del canal.

Ejemplos

Ningún ejemplo hace uso de esta función.

9.5.6 e_dma_wait()

Definición

```
void e_dma_wait(
     e_dma_id_t   chan
);
```

Descripción

Detiene la ejecución del programa y espera mientras canal de DMA chan está ocupado.

Se puede utilizar esta función tras iniciar una transacción para asegurarse de que no se

Valor devuelto

Ninguno.

Ejemplos

Esta función se utiliza en los ejemplo que utilizan DMA después de iniciar la trasferencia con la función *e_dma_start()*. De esta forma, el programa no continúa hasta que la transferencia ha terminado.

9.5.7 e_dma_set_desc()

Definición

```
void e_dma_set_desc(
      e_dma_id_t    chan,
      unsigned      config,
      e_dma_desc_t  *next_desc,
      unsigned      stride_i_src,
      unsigned      stride_i_dst,
      unsigned      count_i,
      unsigned      count_o,
      unsigned      stride_o_src,
      unsigned      stride_o_dst,
      void          *addr_src,
      void          *addr_dst,
      e_dma_desc_t  *descriptor
);
```

Descripción

La descripción de esta función es la más compleja que se puede encontrar en este manual, debido a su extensión y a que el tema no es trivial. Como podrá comprobar, no se trata de una traducción del manual oficial.

Esta función establece en `descriptor` los valores que se utilizarán en una transacción DMA. El canal DMA viene sera `chan`. Si se indica que al acabar esta transacción debe ejecutarse otra transacción, de forma encadenada, el descriptor de dicha transacción será `next_desc`.

Podemos imaginar una transacción DMA como una copia desde el origen al destino utilizando un bucle parecido a este:

```
for(i=0; i<count_i; i++){
      addr_dst[i * stride_i_dst] = addr_src[i * stride_i_src];
}
```

En este bucle se especifican el destino (`addr_dst`) y origen (`addr_src`) de los datos, cuántos datos deben copiarse (`count_i`), y cuántos bytes debe saltarse (`stride_i_dst` y `stride_i_src`) hasta el siguiente dato, ya que no ocupa lo mismo un carácter (1 byte) que un número entero (4 bytes)

Si lo dicho anteriormente queda claro, es fácil entender la mayor parte de los ejemplos `/dma_start`. En estos ejemplos el *host* coloca 10 números en la

memoria externa (compartida) y el núcleo los copia a la memoria local mediante una transacción DMA.

En los ejemplos siempre se utiliza el canal E_DMA_0.

El valor de `config` es el resultado de hacer un OR entre todas las constantes de configuración que necesitamos para especificar qué tipo de transacción DMA es esta. Las constantes y su significado son los siguientes:

- **E_DMA_ENABLE** – Enciende el canal DMA `chan`.
- **E_DMA_MASTER** - Indica que el canal trabaje en modo maestro. No utilizar este modo equivale a indicar que trabaje en modo esclavo.
- **E_DMA_CHAIN** – Indica quede debe encadenar esta transacción con otra utilizando para ello el descriptor `next_desc`.
- **E_DMA_STARTUP** – Indica que se inicie la transferencia inmediatamente, sin necesidad de utilizar a la función *e_dma_start()*.
- **E_DMA_IRQEN** - Permite que se produzca una interrupción al finalizar una transferencia DMA. En el caso de interrupciones encadenadas, la interrupción se establece antes que se cargue el siguiente descriptor.
- **E_DMA_BYTE** – Indica que el tamaño de los datos es 1 byte.
- **E_DMA_HWORD** – El tamaño de los datos es half word (2 byte).
- **E_DMA_WORD** – El tamaño de los datos es word (4 byte).
- **E_DMA_DWORD** – El tamaño de los datos es double word (8 byte).

De esta lista se han quitado los modos experimentales, pero se ha utilizado uno en los ejemplos y es lógico comentar su significado.

- **E_DMA_MSGMODE** – Cuando una transición acaba, se modifica un registro DMA para indicarlo. Por lo tanto, para saber si una transacción ha acabado es necesario chequear constantemente este registro. Este modo experimental permite controlar todo esto a través de la función *e_dma_wait()*, de manera que no es necesario chequear ningún registro. Existe un ejemplo /dma_start_master_v2 que no utiliza este modo y donde se puede ver cómo chequear la finalización de una transacción.

Como no queremos encadenar ninguna otra transacción, el valor para `next_desc` es 0x0000, que es como no pasarle nada.

Como son 10 números utilizamos para el contador 0x000A (10 en hexadecimal). El segundo contador debe ser positivo, por eso se le da 0x0001.

El paso utilizado, tanto en el origen como en el destino es 0x0004, ya que los números enteros ocupan 4 bytes.

Por último, como el buffer en la memoria externa (compartida) apunta a dicha memoria y el array local de números apuntando a la memoria local, ya tenemos las direcciones de origen y destino para la transacción.

Valor devuelto

Ninguno.

Ejemplos

Todos los ejemplos DMA hacen uso de esta función para construir su descriptor de transacciones.

9.6 Funciones para Mutex y Barrera

Los mutex (abreviatura de *mut*ual *ex*clusion) son objetos que permiten bloquear un recurso compartido, garantizando el acceso al mismo de forma exclusiva. Cuando se requiere el acceso a un recurso compartido, primero se comprueba el mutex asociado al mismo. Si el mutex está libre, el recurso también lo está. El proceso que adquiera el control del mutex tiene garantiza su acceso al recurso compartido mientras no libere el mutex.

Las barrerras se utilizan para sincronizar hilos de ejecución paralelos. Si se carga un programa con una barrera en varios núcleos, cuando un núcleo llega a una barrera se detiene y espera a que el resto de núcleos lleguen a la misma barrera. Entonces todos los núcleos continúan con su ejecución.

Para los mutex y barreras se han definido los siguientes tipos:

```
typedef int e_mutex_t;

typedef int e_mutexattr_t;

typedef char e_barrier_t;
```

Las funciones para mutex son:

- e_mutex_init()
- e_mutex_lock()
- e_mutex_trylock()
- e_mutex_unlock()

Las funciones para barreras son:

- e_barrier_init()
- e_barrier()

9.6.1 e_mutex_init()

Definición

```
void e_mutex_init(
     unsigned        row,
     unsigned        col,
     e_mutex_t       *mutex,
     e_mutexattr_t   *attr
);
```

Descripción

Esta función inicializa `mutex` en el núcleo (`row`, `col`) del grupo de trabajo. Una vez inicializado el estado de `mutex` es libre.

El atributo de inicialización, especificado por `attr`, está reservado para uso futuro. Cuando se use esta función, utilice `NULL`.

Valor devuelto

Ninguno.

Ejemplos

`/mutex` – En este ejemplo varios núcleos se incrementan un valor almacenado en la memoria externa (compartida). Esta función se utiliza para inicializar mutex en el propio núcleo.

9.6.2 e_mutex_lock()

Definición

```
void e_mutex_lock(
     unsigned   row,
     unsigned   col,
     e_mutex_t  *mutex
);
```

Descripción

Esta función intenta adquirir el control de `mutex` en el núcleo (`row`, `col`) del grupo de trabajo. Si el control del `mutex` ha sido adquirido por otro núcleo, detiene su ejecución hasta adquirir el control de mutex.

Valor devuelto

Ninguno.

Ejemplos

`/mutex` – En este ejemplo varios núcleos se incrementan un valor almacenado en la memoria externa (compartida). Esta función se utiliza para inicializar adquirir o bloquear mutex, de forma que se tenga acceso exclusivo al valor que se quiere incrementar.

Si eliminamos esta función, todos los núcleos intentan acceder a la vez al mismo recurso. Esto provoca que varios núcleos puedan leer el mismo valor, intentando posteriormente escribir el mismo valor incrementado. Como consecuencia, el valor final es menor al esperado. Pruebe a comentar esta función en el ejemplo para ver sus consecuencias.

9.6.3 e_mutex_trylock()

Definición

```
unsigned e_mutex_trylock(
      unsigned row,
      unsigned col,
      e_mutex_t *mutex
);
```

Descripción

Esta función intenta adquirir el control `mutex` en el núcleo (`row`, `col`) del grupo de trabajo. Si el control de `mutex` ha sido adquirido por otro núcleo, no se queda a la espera de adquirir el control de `mutex`, continua su ejecución.

Valor devuelto

Si tiene éxito al adquirir el control de `mutex`, la función devuelve 0. En otro caso, devuelve el ID del núcleo que lo controla.

Ejemplos

Ningún ejemplo hace uso de esta función.

Es posible modificar el ejemplo `/mutex` para incrementar el valor sólo cuando esta función no devuelva 0, es decir, cuando no se consiga obtener el control del mutex. De esa forma, contabilizaríamos cuántas de veces se intenta sin éxito obtener el control del mutex. Pero cuidado, estaríamos contabilizándolo utilizando un recurso compartido sin control, por lo que el resultado real será probablemente mayor.

9.6.4 e_mutex_unlock()

Definición

```
void e_mutex_unlock(
     unsigned   row,
     unsigned   col,
     e_mutex_t  *mutex
);
```

Descripción

La función liberar el control de `mutex` en el núcleo (`row`, `col`) del grupo de trabajo.

Valor devuelto

Si tiene éxito, la función devuelve 0. En otro caso, devuelve un valor distinto.

Ejemplos

`/mutex` – En este ejemplo varios núcleos se incrementan un valor almacenado en la memoria externa (compartida). Esta función se utiliza para liberar el mutex, de forma que otros núcleos tengan acceso exclusivo al valor que se quiere incrementar.

Si eliminamos esta función, los demás núcleos quedan bloqueados a la espera de que mutex sea liberado, y lo que es más curioso, el propio núcleo queda bloqueado en la siguiente iteración del bucle, ya que intentará adquirir un recurso bloqueado por el mismo.

9.6.5 e_barrier_init()

Definición

```
void e_barrier_init(
     volatile e_barrier_t   bar_array[],
     e_barrier_t            *tgt_bar_array[]
);
```

Descripción

Esta función inicializa una barrera en el grupo de trabajo. Los parámetros `bar_array` y `tgt_bar_array` son arrays de un tamaño igual al número de núcleos en el grupo de trabajo.

La barrera es la misma para todos los núcleos en el grupo de trabajo, por lo que se debe tener cuidado al colocar la llamada *e_barrier()*, para evitar las condiciones de estancamiento.

Valor devuelto

Ninguno.

Ejemplos

`/barrier` – En este ejemplo se van iniciando uno a uno todos los núcleos. El programa en el núcleo utiliza una barrera, de forma que, aunque cada núcleo empieza en un momento diferente, hasta que todos no llegan a la barrera, los núcleos no pueden terminar con la ejecución del programa. Esta función se utiliza para inicializar la barrera.

9.6.6 e_barrier();

Definición

```
void e_barrier(
     volatile e_barrier_t  *bar_array,
     e_barrier_t           *tgt_bar_array[]
);
```

Descripción

Esta función establece una barrera (un punto de sincronización) en el grupo de trabajo. Cuando el programa alcanza esta barrera se detiene y esperar hasta que todos los núcleos del grupo de trabajo alcanzan esta barrera.

Los parámetros `bar_array` y `tgt_bar_array` son arrays de un tamaño igual al número de núcleos en el grupo de trabajo, y deben ser inicializados por *e_barrier_init()*.

La barrera es la misma para todos los núcleos en el grupo de trabajo, por lo que se debe tener cuidado al colocar la llamada *e_barrier()*, para evitar las condiciones de estancamiento.

Valor devuelto

Ninguno.

Ejemplos

`/barrier` – En este ejemplo se van iniciando uno a uno todos los núcleos. El programa del núcleo utiliza una barrera, de forma que, aunque cada uno empieza en un momento diferente, hasta que todos no llegan a la barrera no terminan. Esta función se utiliza para detener la ejecución del núcleo hasta que lleguen todos los núcleos a la barrera.

9.7 Core ID and Workgroup Functions

En el capítulo dedicado a la arquitectura se ha explicado que cada núcleo está situado en un lugar de la memoria, de cuya dirección podemos deducir un identificador denominado Core ID, y de este identificador se puede deducir las coordenadas del núcleo, pero no las relativas al chip, como (0, 0).

Las funciones relacionadas con los Core ID y los grupos de trabajo son:

- e_get_coreid()
- e_get_global_address()
- e_coreid_from_coords()
- e_coords_from_coreid()
- e_is_on_core()
- e_neighbor_id()

Y a continuación los tipos de datos definidos para ellos:

```
typedef unsigned int e_coreid_t;
#define E_SELF

typedef enum
{
     // neighboring cores wrap topology
     E_GROUP_WRAP,  // all chip cores form a ring
     E_ROW_WRAP,    // core rows form rings
     E_COL_WRAP,    // core columns form rings

     // neighboring cores direction
     E_NEXT_CORE,   // neighbor core with the next coreID
     E_PREV_CORE,   // neighbor core with the prev coreID
} e_coreid_wrap_t;
```

```
typedef enum {
     E_E16G301,
     E_E64G401,
} e_chiptype_t;

typedef struct {
     e_chiptype_t chiptype;
     e_coreid_t     group_id;
     unsigned  group_row;
     unsigned  group_col;
     unsigned  group_rows;
     unsigned  group_cols;
     unsigned  core_row;
     unsigned  core_col;
     unsigned  alignment_padding;
} e_group_config_t;

typedef struct {
     unsigned  base;
} e_emem_config_t;
```

9.7.1 e_get_coreid()

Definición

```
e_coreid_t e_get_coreid(void);
```

Descripción

Obtiene el Core ID del núcleo.

Valor devuelto

Devuelve un valor de 12 bits que representa el Core ID del núcleo.

Ejemplos

`/mutex` – En este ejemplo el host carga un programa en varios núcleos que incrementan el valor de una variable en la memoria externa (compartida). En este ejemplo se usa esta función para obtener el ID del núcleo, necesario para calcular sus coordenadas, necesarias para iniciar el mutex.

`/barrier` – En este ejemplo se van iniciando uno a uno todos los núcleos. El programa del núcleo utiliza una barrera, de forma que, aunque cada uno empieza en un momento diferente, hasta que todos no llegan a la barrera no terminan. En este ejemplo se usa esta función para obtener el ID del núcleo, necesario para calcular sus coordenadas, y con estas, su número dentro del grupo de trabajo.

9.7.2 e_get_global_address ()

Definición

```
void *e_get_global_address(
     unsigned     row,
     unsigned     col,
     const void   *ptr
);
```

Descripción

Esta función transforma el puntero local `ptr` en la dirección correspondiente en un núcleo vecino (`row`, `col`) del grupo de trabajo.

Hay que tener en cuenta que, cuando `ptr` apuntan a una dirección global, la función devuelve una versión no modificada de `ptr`.

Si `row` o `col` son `E_SELF`, o son iguales a las coordenadas del núcleo que hace la llamada, la función calcula la dirección global que corresponde a esa dirección local, es decir, la dirección con la que otro núcleo apuntaría a este.

Valor devuelto

Devuelve un valor de 32 bits que representan una dirección global.

Ejemplos

`/cajero` – En este ejemplo se simula un cajero automático del que se quiere sacar un importe determinado. Para indicar a cada núcleo cuánto dinero queda por devolver, se utiliza su banco de memoria. Esta función la utilizan los núcleos para determinar la dirección del banco de memoria del siguiente núcleo, y así poder indicarle cuánto dinero queda por devolver. Se utiliza otro banco de memoria para indicar que puede empezar a realizar los cálculos.

9.7.3 e_coreid_from_coords()

Definición

```
e_coreid_t e_coreid_from_coords(
     unsigned   row,
     unsigned   col
);
```

Descripción

Obtiene el Core ID del núcleo (row, col) del grupo de trabajo.

Valor devuelto

Devuelve un valor de 12 bits que corresponde a un Core ID

Ejemplos

Ningún ejemplo hace uso de esta función.

Si quiere probar esta función puede utilizar el ejemplo `/inc_emem_buff` para devolver el ID de un núcleo en lugar del valor incrementado almacenado en la memoria extendida (compartida).

9.7.4 e_coords_from_coreid()

Definición

```
void e_coords_from_coreid(
     e_coreid_t   coreid,
     unsigned     *row,
     unsigned     *col
);
```

Descripción

Calcula las coordenadas (`row`, `col`) del núcleo cuyo Core ID es `coreid` dentro del grupo de trabajo.

Tenga en cuenta que la función no comprueba que el Core ID pertenece a un núcleo del grupo de trabajo. Si el núcleo no pertenece al grupo de trabajo el valor devuelto puede ser negativo o superior al tamaño del grupo.

Valor devuelto

Ninguno.

Ejemplos

`/mutex` – En este ejemplo el host carga un programa en varios núcleos que incrementan el valor de una variable en la memoria externa (compartida). En este ejemplo se usa esta función para obtener las coordenadas del núcleo, necesarias para iniciar el mutex.

`/barrier` – En este ejemplo se van iniciando uno a uno todos los núcleos. El programa del núcleo utiliza una barrera, de forma que, aunque cada uno empieza en un momento diferente, hasta que todos no llegan a la barrera no terminan. En este ejemplo se usa esta función para obtener las coordenadas, necesarias para calcular su número dentro del grupo de trabajo.

9.7.5 e_is_on_core()

Definición

```
e_bool_t e_is_on_core(
     const void *ptr
);
```

Descripción

Esta función comprueba si la dirección apuntada por `ptr` está dentro del espacio de memoria correspondiente al núcleo que hace la llamada.

Valor devuelto

Devuelve `E_TRUE` si la dirección está dentro del espacio de memoria correspondiente al núcleo que hace la llamada y `E_FALSE` en caso contrario.

Ejemplos

Ningún ejemplo hace uso de esta función. Puede probar esta función en el ejemplo `/cajero`, donde se calcula a partir de un puntero que apunta a la memoria local, la dirección a la memoria local de otro núcleo.

9.7.6 e_neighbor_id()

Definición

```
void e_neighbor_id(
     e_coreid_wrap_t   dir,
     e_coreid_wrap_t   wrap,
     unsigned          *row,
     unsigned          *col
);
```

Descripción

Esta función solo se puede utilizar cuando el número de núcleos del grupo de trabajo es potencias de 2, es decir, 2^n: 1, 2, 4, 8, 16…

Esta función calcula las coordenadas (`row`, `col`) del núcleo vecino. El parámetro `dir` (`E_NEXT_CORE`, `E_PREV_CORE`) indica si se quieren el vecino siguiente o el anterior. Los vecinos dependen del valor de `wrap`, que indica el tipo de encadenamiento de los núcleos, que puede ser:

- **E_COL_WRAP** – Recorre las columnas de arriba a abajo.
- **E_ROW_WRAP** – Recorre las filas de izquierda a derecha.
- **E_GROUP_WRAP** – Recorre las filas de izquierda a derechas, y llegado al final de la fila, pasa a la fila siguiente.

Las ilustración 9-1 muestra el sentido en el que avanzan los núcleos utilizando `E_COL_WRAP` y `E_ROW_WARP`, y la ilustración 9-2 al utilizar `E_GROUP_WRAP`.

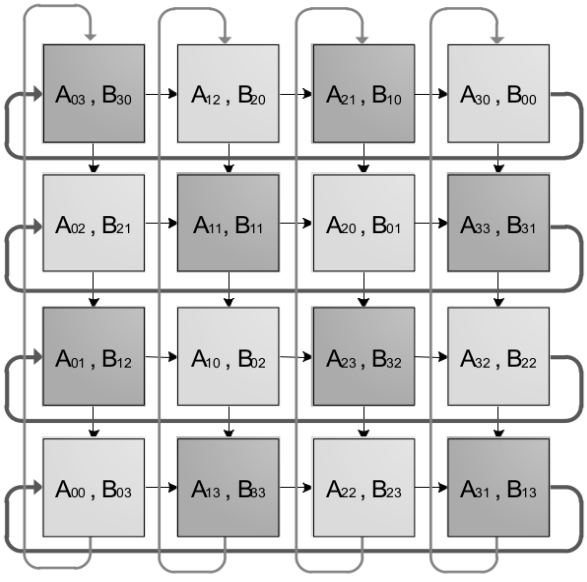


Ilustración 9-20. E_COL_WRAP y E_ROW_WRAP

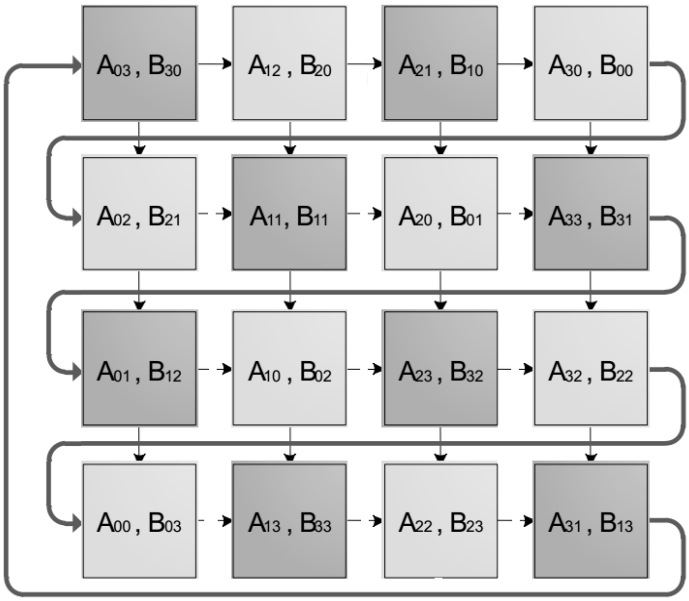


Ilustración 9-21. E_GROUP_WRAP

Valor devuelto

Ninguno.

Ejemplos

/cajero – En este ejemplo se simula un cajero automático del que se quiere sacar un importe determinado. Esta función la utilizan los núcleos para determinar las coordenadas del siguiente núcleo. En el ejemplo se utiliza una configuración E_GROUP_WRAP.

10 CONCLUSIONES

En este capítulo expreso las conclusiones de este trabajo fin de grado, esto es, experiencia personal, líneas de investigación futuras y mi opinión acerca del proyecto Parallella.

10.1 Experiencia personal

Lo primero que destacaría sería la incomodidad de no contar con un entorno de desarrollo para programar. En el foro oficial se puede ver que hay gente pidiendo ayuda sobre cómo configurar Eclipse. Hay quien al ver las limitaciones de Eclipse respecto a Parallella-16, que solo es compatible con los prototipos y que Adapteva ha dejado de proporcionar Eclipse, pasan directamente a programar, compilar y ejecutar desde la línea de comandos.

En mi caso, como se puede ver en la ilustración 10-1, opté por utilizar Emacs en Parallella-16, el único editor de texto en modo visual que venía instalado y resaltaba la sintaxis. Con el tiempo pasé a utilizarlo en modo texto. Aunque he llegado a apreciar sus muchas virtudes, hay que ser consciente de que hoy en día, los programadores esperan contar con algo como Eclipse o NetBeans.

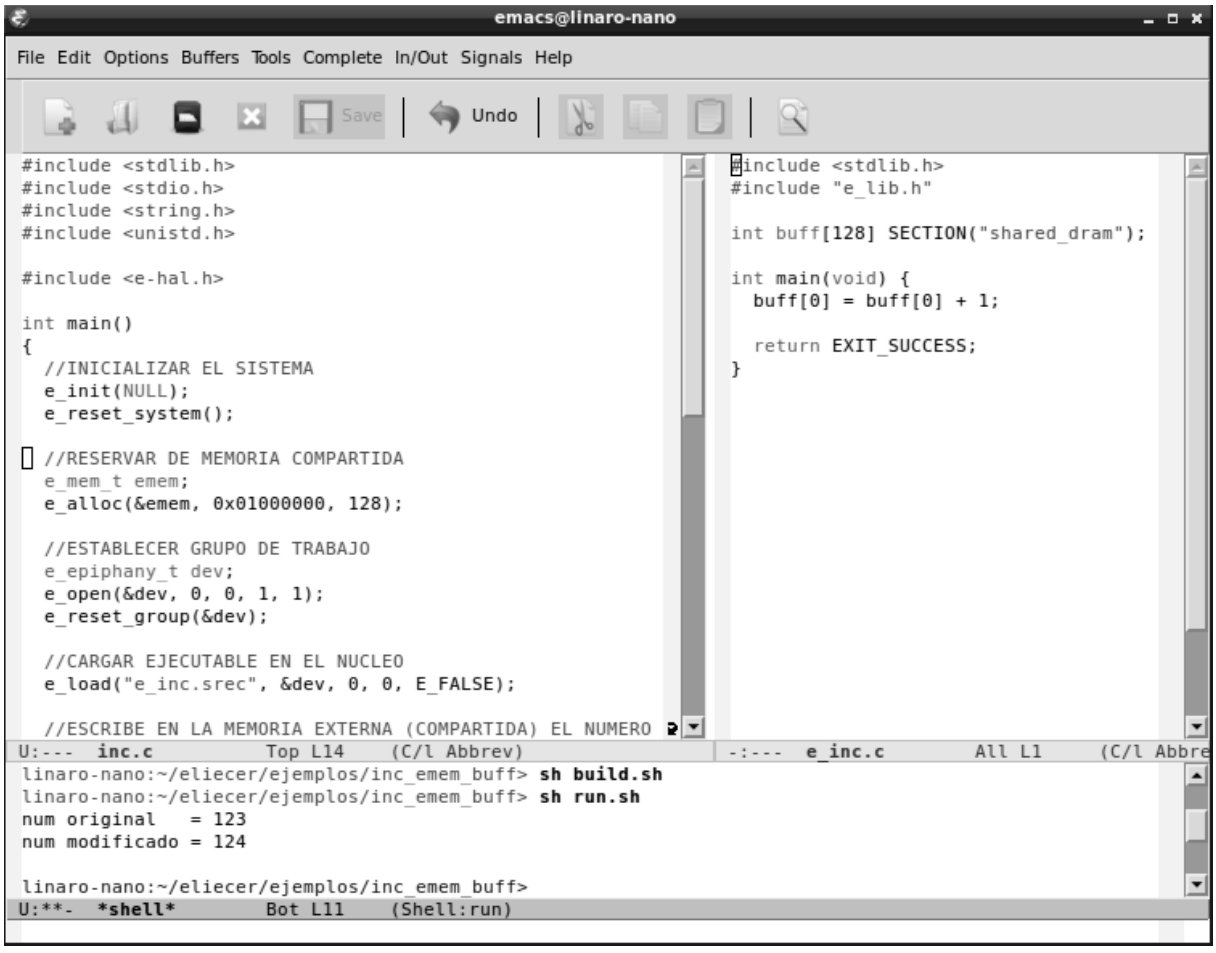


Ilustración 10-22. Emacs (modo visual) como entorno de programación

Ha sido difícil investigar, averiguar y probar cómo utilizar las funciones de las librerías, y eso que están documentadas. Ofrecer un ejemplo simple es fundamental para entender la función y el contexto en el que se utiliza. Es lo que he intentado con el conjunto de ejemplos que he creado.

El hecho de que el programador de Parallella-16 no pueda ignorar su arquitectura, y que deba conocer por ejemplo los detalles de su memoria, son incomodidades que se resuelven acudiendo a sus manuales, pero deducir de ellos que para reservar memoria compartida con una función se deben saltar 16777216 bytes, es algo que solo te resuelve alguien con muchos conocimientos en el foro oficial. Yo lo explico cuando describo la función y en el capítulo *Hello World*, que es como una introducción a la programación en esta arquitectura. Es algo que todo el mundo debería conocer si piensa programar en Parallella-16.

Por último, me alegro de haber realizado este trabajo fin de grado, me ha puesto a prueba. En demasiadas ocasiones he pensado que estaba a punto de terminar, y me encontraba con un nuevo problema. Para investigar hay que tener tiempo, paciencia, perseverar y no dar por hecho nada. Ojalá tanto esfuerzo haya merecido la pena.

10.2 Líneas de investigación futuras

Son muchas las cosas que no he tocado en este trabajo fin de grado:

- **Depuración**. No he probado ni documentado la depuración en Parallella-16. Sin la depuración, los núcleos son auténticas cajas negras donde uno no sabe que está ocurriendo. Por otro lado, la depuración de un programa ejecutado en 16 núcleos requiere de un enfoque distinto. Alguien podría completar este trabajo añadiendo algo al respecto, el siguiente paso de Adapteva es sacar Parallella-16, con 64 núcleos. Su enfoque de la depuración no será el tradicional.

- **Varias Parallella-16**. En la fecha en la que se escribieron están líneas no era posible adquirir nuevas Parallella-16 por estar agotadas. Yo solo he contado con una. El potencial que tiene poder acoplar varios de estos ordenadores es digno de estudio y de un trabajo fin de grado.

- **Librerías adicionales**. Como forma de facilitar la programación se podría aportar nuevas librerías. En el foro hay quien propone una que permita compartir datos de forma más sencilla entre el host (procesador ARM) y los núcleos.

- **Video-tutoriales**. A veces resulta más fácil explicar algo mostrándolo. No existen videos relacionados con la programación de Parallella-16. Los vídeos relacionados con este ordenador son tan escasos que resulta difícil imaginar que esto no se le haya ocurrido a los responsables de Adapteva.

- **Comparativas**. Empiezan a verse Raspberry Pi ejecutándose en paralelo. Podría realizarse una comparativa a cerca del rendimiento y la potencia de ambos como alternativa a los superordenadores tradicionales.

- **Parallella-16 como base de un proyecto**. El tamaño, el peso, la potencia consumida y su precio lo hacen apto para un montón de proyectos donde estas variables dificultan su desarrollo. Pienso en cosas como la robótica, los drones, el espacio, etc.

10.3 Mi opinión sobre el proyecto Parallella

Creo que Adapteva con Parallella-16 ha dado un gran paso, cualquier laboratorio puede contar con un ordenador con 64 núcleos por el precio de un ordenador de sobre mesa. Pero Adapteva se ha olvidado de una cosa, poner las cosas fáciles.

Parallella-16 es como un filón de oro perdido en la montaña, me da la sensación que la comunidad de investigadores desconoce que existe, o ha sido incapaz de hacer algo con él. Por alguna de estas razones, o ambas, no se ha visto aún a nadie hacer nada novedoso o basado en este ordenador.

Si yo fuera el director de Adapteva lo primero que haría sería intentar vender a cada laboratorio de investigación de las universidades de Estados Unidos 4 Parallella-16, su precio es ridículo, y los beneficios serían altos. Estoy seguro que con este dinero podría financiar los nuevos modelos y pagar a gente que documentara adecuadamente su uso y creara un conjunto de videos tutoriales y promocionales que colgados en Internet darían sus frutos.

Con la documentación actual, que está desactualizada, algo que ellos mismos reconocen, con 4 videos en YouTube, con un foro en el que se ve que la gente está muy perdida, y un canal de IRC que ha sido declarado ya en alguna ocasión congelado por no haber actividad en él durante casi 2 semanas, por muy buena que sea Parallella-16, el proyecto Parallella está abocado al fracaso.

11 ÍNDICE DE ILUSTRACIONES

Ilustración 2-1. Evolución prevista de la arquitectura de ordenadores.....14

Ilustración 2-2. Parallella-16 sobre la palma de una mano.....16

Ilustración 2-3. Componentes de Parallella-16.....16

Ilustración 2-4. Parte inferior de Parallella-16.....17

Ilustración 2-5. Conexión entre componentes de Parallella-16.....17

Ilustración 4-6. Arquitectura multi-núcleo Epiphany.....25

Ilustración 4-7. Memoria de la arquitectura Epiphany.....26

Ilustración 4-8. Malla 2D de cuatro Parallellas-16.....27

Ilustración 6-9. Creación de un proyecto en Eclipse.....39

Ilustración 6-10. Indicando cuantos núcleos utilizar en Eclipse.....40

Ilustración 6-11. Propiedades del proyecto en Eclipse.....41

Ilustración 6-12. Ejecución del e-server.....42

Ilustración 6-13. Eclipse en modo depuración ejecutando hello-world.....43

Ilustración 7-14. Configuración PC y Parallella-16.....46

Ilustración 7-15. Relé USB con los cables de alimentación.....47

Ilustración 7-16. Parallella-16 con un disipador de calor.....47

Ilustración 7-17. Utilizando Emacs (modo texto) desde PuTTY.....48

Ilustración 7-18. Parallella-16 desde Windows 7 con Conexión a Escritorio remoto. 49

Ilustración 7-19. Parallella-16 desde Ubuntu con Remmina.....50

Ilustración 9-20. E_COL_WRAP y E_ROW_WRAP.....130

Ilustración 9-21. E_GROUP_WRAP.....131

Ilustración 10-22. Emacs (modo visual) como entorno de programación.....133

12 ÍNDICE DE TABLAS

Tabla 2-1. Especificaciones técnicas de Parallella-16.....15

13 REFERENCIAS

En este capítulo se listan las referencias, todas fuentes electrónicas, consultadas para realizar este trabajo fin de grado.

- Adapteva. **Epiphany Architecture Reference [REV 14.03.11]**. Disponible en <http://www.adapteva.com/docs/epiphany_arch_ref.pdf>.
- Adapteva. **Epiphany SDK Reference [REV 5.13.09.10]**. Disponible en <http://www.adapteva.com/docs/epiphany_sdk_ref.pdf>.
- Adapteva. **Epiphany-III (E16G301) Datasheet [REV 14.03.11]**. Disponible en <http://www.adapteva.com/docs/e16g301_datasheet.pdf>.
- Adapteva. **Epiphany-IV (E64G401) Datasheet [REV 14.03.11]**. Disponible en <http://www.adapteva.com/docs/e64g401_datasheet.pdf>.
- Adapteva. **Parallella Reference Manual [REV 14.09.09]**. Disponible en <http://www.adapteva.com/docs/e64g401_datasheet.pdf>.
- Colaboradores de Wikipedia. **Raspberry Pi**. Wikipedia, La enciclopedia libre, 2015 [fecha de consulta: 21 de enero del 2015]. Disponible en <http://es.wikipedia.org/w/index.php?title=Raspberry_Pi>.
- Colaboradores de Wikipedia. **Placa computadora**. Wikipedia, La enciclopedia libre, 2014 [fecha de consulta: 21 de enero del 2015]. Disponible en <http://es.wikipedia.org/w/index.php?title=Placa_computadora>.
- Colaboradores de Wikipedia. **Adapteva**. Wikipedia, La enciclopedia libre, 2014 [fecha de consulta: 21 de enero del 2015]. Disponible en < http://en.wikipedia.org/w/index.php?title=Adapteva>.
- Colaboradores de Wikipedia. **SIMD**. Wikipedia, La enciclopedia libre, 2014 [fecha de consulta: 21 de enero del 2015]. Disponible en <http://es.wikipedia.org/w/index.php?title=SIMD>.
- Colaboradores de Wikipedia. **MIMD**. Wikipedia, La enciclopedia libre, 2013 [fecha de consulta: 21 de enero del 2015]. Disponible en <http://es.wikipedia.org/w/index.php?title=MIMD>.
- Colaboradores de Wikipedia. **SPMD**. Wikipedia, La enciclopedia libre, 2014 [fecha de consulta: 21 de enero del 2015]. Disponible en <http://es.wikipedia.org/w/index.php?title=SPMD>.
- Dr. Manuel Sánchez y Bernal. **Arquitectura-SIMD**. Universidad de Santiago de Chile, 2015 [fecha de consulta: 21 de enero del 2015]. Disponible en <http://msanchez.usach.cl/Icc/Arquitectura-SIMD.pdf>.

ANEXO: CONTENIDO DEL CD

El CD que acompaña este trabajo fin de grado tiene el siguiente contenido.

13.1 Memoria

La carpeta `/memoria` contiene una versión de esta memoria en **formato DOCX** creada utilizando Microsoft Word 2010 y otra en **formato PDF**.

La versión PDF no se ha generado desde Word porque no crea marcadores de navegación a partir del índice. Los manuales oficiales de Parallella-16 contienen estos marcadores y son una forma fantástica de navegar por el documento, mucho mejor que ir al índice. He querido ofrecer esta misma comodidad a quien acabe en una copia de mi proyecto.

Para crear este PDF he creado una copia en formato DOC a partir del DOCX. Después he abierto esta nueva copia con LibreOffice y he hecho algunos cambios, ya que el formato cambia un poco y descuadran las páginas. Finalmente, y desde LibreOffice, he pasado ese documento a PDF y he comprobado que los marcadores aparecían y se podían utilizar sin problemas.

13.2 ESDK

En la carpeta `/esdk` se puede encontrar una copia de los eSDK utilizados en este proyecto:

- **ESDK 5.13.09 para ARM**. La versión utilizada con Parallella-16.
- **ESDK 5.13.07 para x86_64**. La última versión que incluye el entorno de desarrollo basado en Eclipse y la utilizada para realizar el capítulo 6 de esta memoria.

El resto del software utilizado (Ubuntu, Emacs, Adobe Reader, PuTTY, WinSCP, vnc4server) es gratuito, y por esta razón no los he incorporado al CD.

13.3 Ejemplos

En la carpeta `/ejemplos` se encuentran los ejemplos a los que se hace referencia a lo largo de esta memoria.

- **`/address`** - En este ejemplo se crea un grupo de trabajo y un puntero a una zona de memoria. Este ejemplo utiliza las funciones que permiten conocer si un puntero apunta al espacio de memoria del chip o al del grupo de trabajo.

- **`/barrier`** - Este ejemplo utiliza las funciones relacionadas con las barreras. En este ejemplo se van iniciando uno a uno todos los núcleos. El programa del núcleo utiliza una barrera, de manera que, aunque cada uno empieza en un momento diferente, hasta que todos no llegan a la barrera no terminan.

- **`/cajero`** - Este ejemplo simula un cajero automático, en el sentido de que calcula cuantos billetes y de qué valor debe devolver si se le pide un importe determinado. Este ejemplo hace uso de las funciones que permiten conocer los núcleos vecinos.

- **`/core_coords`** - En este ejemplo se listan todos los núcleos de un grupo de trabajo junto a sus coordenadas. Este ejemplo demuestra cómo obtener en el *host*, las coordenadas de un núcleo conociendo su número.

- **`/core_number`** - En este ejemplo se crea un grupo de trabajo y se saca por pantalla una representación visual del chip (16 núcleos dispuestos 4x4). Este ejemplo demuestra cómo obtener en el *host*, el número de un núcleo conociendo sus coordenadas.

- Hay 4 ejemplos donde los núcleos utilizan las funciones DMA para obtener una lista de 10 números para sumarlos.

    - **`/dma_copy`** - Esta versión utiliza la función *e_dma_copy()*, esta función no necesita declarar un descriptor de transacción.

    - **`/dma_start_master`** - Esta versión utiliza un descriptor con los modos `MASTER` y `MSGMODE`.

    - **`/dma_start_master_v2`** - Esta versión utiliza un descriptor solo con el modo `MASTER`.

    - **`/dma_start_slave`** - Esta versión utiliza un descriptor sin el modo `MASTER`, lo que significa implícitamente que trabaja en modo `SLAVE`.

- **`/halt`** – La función *e_halt()* sirve para detener un núcleo. En este ejemplo un núcleo incremente indefinidamente un número almacenado en su banco de memoria. Se utiliza esta función para detener el núcleo antes de leer su valor.

- Hay 3 ejemplos que demuestran cómo pasar/recibir datos entre host y núcleo: Utilizando los bancos de memoria de los núcleos o la memoria compartida. El ejemplo pasa un número y un núcleo lo incrementan.

    - **`/inc_bank`** – Utiliza el banco de memoria del núcleo.

    - **`/inc_emem_buff`** – Accede a la memoria compartida con un buffer.

    - **`/inc_emem_func`** – Accede a la memoria compartida con una función.

- **`/info`** – Muestra como consultar los datos de la plataforma en la que se ejecuta el programa: Versión del chip, cuántos chips tiene, número de filas y columnas, coordenadas del primer núcleo.
- **`/mutex`** – En este ejemplo se muestra cómo los núcleos pueden declarar y utilizar mutex. En el ejemplo varios núcleos intentan incrementar a la vez un número. Para evitar problemas de concurrencia se utiliza el mutex.
- **`/timers`** – En este ejemplo se utilizan las funciones relacionadas con los contadores internos de los núcleos. El ejemplo calcula el tiempo transcurrido por una operación que realiza el núcleo.
- Hay 3 ejemplos que demuestran cómo recibir datos en el host desde los núcleos: Utilizando los bancos de memoria y los registros de los núcleos, o la memoria compartida. El ejemplo devuelve información sobre un núcleo.
    - /whoiam_bank – Utilizando el banco de memoria del núcleo.
    - /whoiam_emem – Utilizando la memoria externa (compartida).
    - /whoiam_reg – Utilizando los registros del núcleo.